2. Resource script: The resource script is an ASCII file that generally has the extension .RC. This file contains definitions of menus, dialog boxes, string tables, and keyboard accelerators used by the program. The resource script can also reference other files that contain icons, cursors, bitmaps, and fonts in binary form, as well as other read-only data defined by the programmer. When a program is running, Windows loads resources into memory only when they are needed and in most cases can discard them if additional memory space is required.

   SAMPLE.RC, the resource script for the SAMPLE program, is shown in Figure 17-12; it contains only the definition of the menu used in the program.

```
#include "sample.h"

Sample MENU
     BEGIN
          POPUP "&Typeface"
               BEGIN
                    MENUITEM "&Script", IDM_SCRIPT, CHECKED
                    MENUITEM "&Modern", IDM_MODERN
                    MENUITEM "&Roman",  IDM_ROMAN
               END
     END
```

Figure 17-12. The resource script for the SAMPLE program.

3. Header (or *include*) file: This file, with the extension .H, can contain definitions of constants or macros, as is customary in C programming. For Windows programs, the header file also reconciles constants used in both the resource script and the program source-code file. For example, in the SAMPLE.RC resource script, each item in the pop-up menu (*Script*, *Modern*, and *Roman*) also includes an identifier—IDM_SCRIPT, IDM_MODERN, and IDM_ROMAN, respectively. These identifiers are merely numbers that Windows uses to notify the program of the user's selection of a menu item. The same names are used to identify the menu selection in the C source-code file. And, because both the resource compiler and the source-code compiler must have access to these identifiers, the header file is included in both the resource script and the source-code file.

   The header file for the SAMPLE program, SAMPLE.H, is shown in Figure 17-13.

```
#define IDM_SCRIPT 1
#define IDM_MODERN 2
#define IDM_ROMAN  3
```

Figure 17-13. The SAMPLE.H header file.

4. Module-definition file: The module-definition file generally has a .DEF extension. The Windows linker uses this file in creating the executable .EXE file. The module-definition file specifies various attributes of the program's code and data segments, and it lists all imported and exported functions in the source-code file. In large programs that are divided into multiple code segments, the module-definition file allows the programmer to specify different attributes for each code segment.

The module-definition file for the SAMPLE program is named SAMPLE.DEF and is shown in Figure 17-14.

```
NAME           SAMPLE
DESCRIPTION    'Demonstration Windows Program'
STUB           'WINSTUB.EXE'
CODE           MOVABLE
DATA           MOVABLE MULTIPLE
HEAPSIZE       1024
STACKSIZE      4096
EXPORTS        WndProc
```

Figure 17-14. The SAMPLE.DEF module-definition file.

5. Make file: To facilitate construction of the executable file from these different components, Windows programmers often use the MAKE program to compile only those files that have changed since the last time the program was linked. To do this, the programmer first creates an ASCII text file called a make file. By convention, the make file has no extension.

   The make file for the SAMPLE program is named SAMPLE and is shown in Figure 17-15. The programmer can create the SAMPLE.EXE executable file by executing

   ```
   C>MAKE SAMPLE  <Enter>
   ```

   A make file often contains several sections, each beginning with a target filename, followed by a colon and one or more dependent filenames, such as

   ```
   sample.obj : sample.c sample.h
   ```

   If either or both the SAMPLE.C and SAMPLE.H files have a later creation time than SAMPLE.OBJ, then MAKE runs the program or programs listed immediately below. In the case of the SAMPLE make file, the program is the C compiler, and it compiles the SAMPLE.C source code:

   ```
   cl -c -Gsw -W2 -Zdp sample.c
   ```

   Thus, if the programmer changes only one of the several files used in the development of SAMPLE, then running MAKE ensures that the executable file is brought up to date, while carrying out only the required steps.

   ```
   sample.obj : sample.c sample.h
        cl -c -Gsw -W2 -Zdp sample.c

   sample.res : sample.rc sample.h
        rc -r sample.rc

   sample.exe : sample.obj sample.def sample.res
        link4 sample, /align:16, /map /line, slibw, sample
        rc sample.res
        mapsym sample
   ```

   Figure 17-15. The make file for the SAMPLE program.

Construction of a Windows program

The make file shows the steps that create a program's .EXE file from the various components:

1. Compiling the source-code file:

```
cl -c -Gsw -W2 -Zdp sample.c
```

   This step uses the CL.EXE C compiler to create a .OBJ object-module file. The command line switches are
   - -c: Compiles the program but does not link it. Windows programs must be linked with Windows' LINK4 linker, rather than with the LINK program the C compiler would normally invoke.
   - -Gsw: Includes two switches, -Gs and -Gw. The -Gs switch removes stack checks from the program. The -Gw switch inserts special prologue and epilogue code in all far functions defined in the program. This special code is required for Windows' memory management.
   - -W2: Compiles with warning level 2. This is the highest warning level, and it causes the compiler to display messages for conditions that may be acceptable in normal C programs but that can cause serious errors in a Windows program.
   - -Zdp: Includes two switches, -Zd and -Zp. The -Zd switch includes line numbers in the .OBJ file — helpful for debugging at the source-code level. The -Zp switch packs structures on byte boundaries. The -Zp switch is required, because data structures used within Windows are in a packed format.
2. Compiling the resource script:

```
rc -r sample.rc
```

   This step runs the resource compiler and converts the ASCII .RC resource script into a binary .RES form. The -r switch indicates that the resource script should be compiled but the resources should not yet be added to the program's .EXE file.
3. Linking the program:

```
link4 sample, /align:16, /map /line, slibw, sample
```

   This step uses the special Windows linker, LINK4. The first parameter listed is the name of the .OBJ file. The /align:16 switch instructs LINK4 to align segments in the .EXE file on 16-byte boundaries. The /map and /line switches cause LINK4 to create a .MAP file that contains program line numbers — again, useful for debugging source code. Next, slibw is a reference to the SLIBW.LIB file, which is an import library that contains module names and ordinal numbers for all Windows functions. The last parameter, sample, is the program's module-definition file, SAMPLE.DEF.
4. Adding the resources to the .EXE file:

```
rc sample.res
```

This step runs the resource compiler a second time, using the compiled resource file, SAMPLE.RES. This time, the resource compiler adds the resources to the .EXE file.

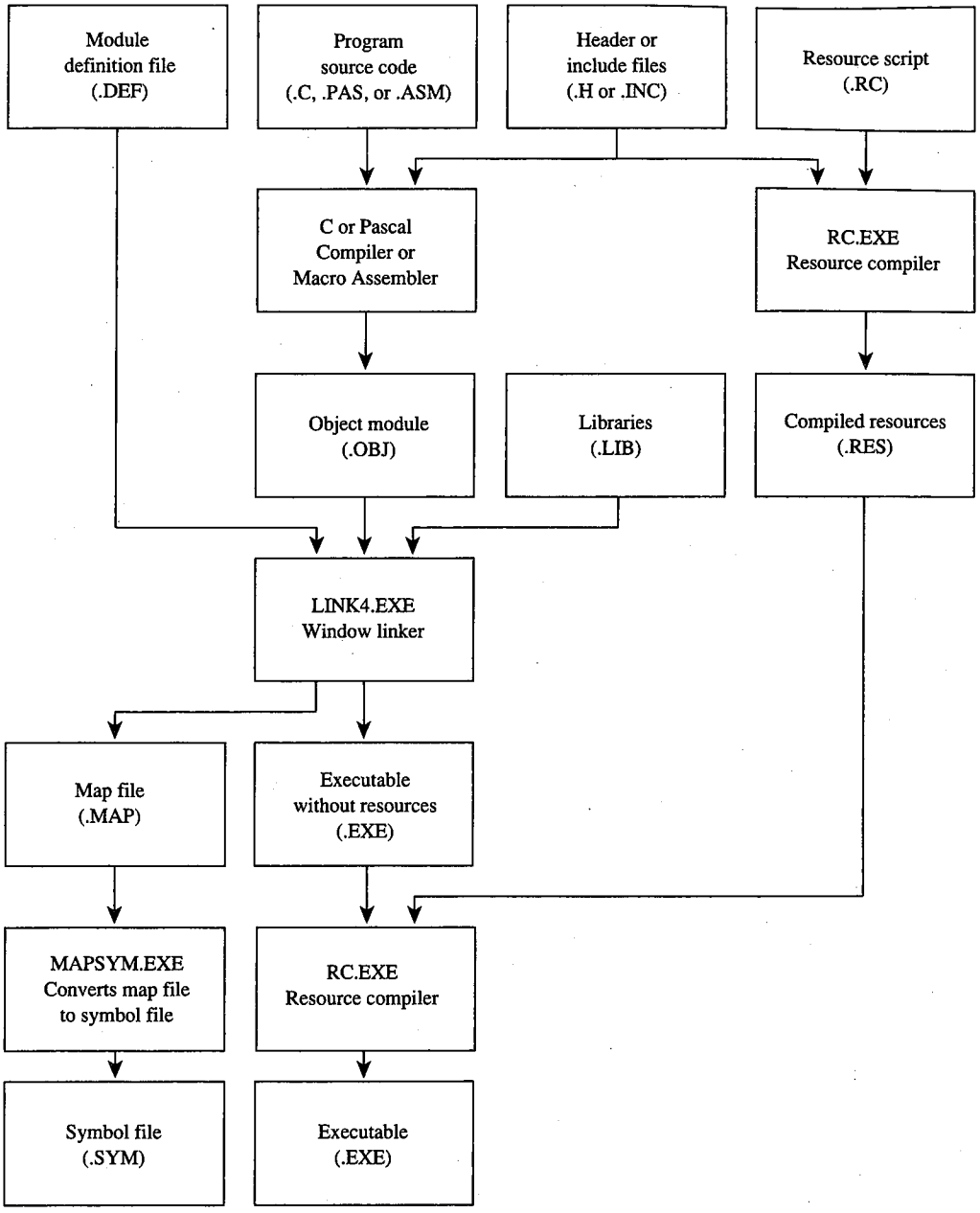


Figure 17-16. A block diagram showing the creation of a Windows .EXE file.

5. Creating a symbol (.SYM) file from the linker's map (.MAP) file:

```
mapsym sample
```

This step is required for symbolic debugging with SYMDEB.

Figure 17-16 on the preceding page shows how the various components of a Windows program fit into the creation of a .EXE file.

Program initialization

The SAMPLE.C program shown in Figure 17-11 contains some code that appears in almost every Windows program. The statement

```
#include <windows.h>
```

appears at the top of every Windows source-code file written in C. The WINDOWS.H file, provided with the Microsoft Windows Software Development Kit, contains templates for all Windows functions, structure definitions, and #define statements for many mnemonic identifiers.

Some of the variable names in SAMPLE.C may look unusual to C programmers because they begin with a prefix notation that denotes the data type of the variable. Windows programmers are encouraged to use this type of notation. Some of the more common prefixes are

Prefix	Data Type
i *or* n	Integer (16-bit signed integer)
w	Word (16-bit unsigned integer)
l	Long (32-bit signed integer)
dw	Doubleword (32-bit unsigned integer)
h	Handle (16-bit unsigned integer)
sz	Null-terminated string
lpsz	Long pointer to null-terminated string
lpfn	Long pointer to a function

The program's entry point (following some startup code) is the WinMain function, which is passed the following parameters: a handle to the current instance of the program (hInstance), a handle to the most recent previous instance of the program (hPrevInstance), a long pointer to the program's command line (lpszCmdLine), and a number (nCmdShow) that indicates whether the program should initially be displayed as a normally sized window or as an icon.

The first job SAMPLE performs in the WinMain function is to register a window class—a structure that describes characteristics of the windows that will be created in the class. These characteristics include background color, the type of cursor to be displayed in the window, the window's initial menu and icon, and the window function (the structure member called lpfnWndProc).

Multiple instances of a program can share the same window class, so SAMPLE registers the window class only for the first instance of the program:

```
if (!hPrevInstance)
          {
          wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
          wndclass.lpfnWndProc   = WndProc ;
          wndclass.cbClsExtra    = 0 ;
          wndclass.cbWndExtra    = 0 ;
          wndclass.hInstance     = hInstance ;
          wndclass.hIcon         = NULL ;
          wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
          wndclass.hbrBackground = GetStockObject (WHITE_BRUSH) ;
          wndclass.lpszMenuName  = szAppName ;
          wndclass.lpszClassName = szAppName ;

          RegisterClass (&wndclass) ;
          }
```

The SAMPLE program then creates a window using the CreateWindow call, displays it to the screen by calling ShowWindow, and updates the client area by calling UpdateWindow:

```
hWnd = CreateWindow (szAppName, "Demonstration Windows Program",
                     WS_OVERLAPPEDWINDOW,
                     (int) CW_USEDEFAULT,0,
                     (int) CW_USEDEFAULT,0,
                     NULL, NULL, hInstance, NULL) ;
ShowWindow (hWnd, nCmdShow) ;
UpdateWindow (hWnd) ;
```

The first parameter to CreateWindow is the name of the window class. The second parameter is the actual text that appears in the window's title bar. The third parameter is the style of the window — in this case, the WINDOWS.H identifier WS_OVERLAPPEDWINDOW. The WS_OVERLAPPEDWINDOW is the most common window style. The fourth through seventh parameters specify the initial position and size of the window. The identifier CW_USEDEFAULT tells Windows to position and size the window according to the default rules.

After creating and displaying a Window, the SAMPLE program enters a piece of code called the message loop:

```
while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
return msg.wParam ;
```

This loop continues to execute until the GetMessage call returns zero. When that happens, the program instance terminates and the memory required for the instance is freed.

The Windows messaging system

Interactive programs written for the normal MS-DOS environment generally obtain user input only from the keyboard, using either an MS-DOS or a ROM BIOS software interrupt to check for keystrokes. When the user types something, such programs act on the keystroke and then return to wait for the next keystroke.

Programs written for Windows, however, can receive user input from a variety of sources, including the keyboard, the mouse, the Windows timer, menus, scroll bars, and controls, such as buttons and edit boxes.

Moreover, a Windows program must be informed of other events occurring within the system. For instance, the user of a Windows program might choose to make its window smaller or larger. Windows must make the program aware of this change so that the program can adjust its screen output to fit the new window size. Thus, for example, if a Windows program is minimized as an icon and the user maximizes its window to fill the full screen, Windows must inform the program that the size of the client area has changed and needs to be re-created.

Windows carries out this job of keeping a program informed of other events through the use of formatted messages. In effect, Windows sends these messages to the program. The Windows program receives and acts upon the messages.

This messaging makes the relationship between Windows and a Windows program much different from the relationship between MS-DOS and an MS-DOS program. Whereas MS-DOS does not provide information until a program requests it through an MS-DOS function call, Windows must continually notify a program of all the events that affect its window.

Window messages can be separated into two major categories: queued and nonqueued.

Queued messages are similar to the keyboard information an MS-DOS program obtains from MS-DOS. When the Windows user presses a key on the keyboard, moves the mouse, or presses one of the mouse buttons, Windows saves information about the event (in the form of a data structure) in the system message queue. Each message is destined for a particular window in a particular instance of a Windows program. Windows therefore determines which window should get the information and then places the message in the instance's own message queue.

A Windows program retrieves information from its queue in the message loop:

```
while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
```

The *msg* variable is a structure. During the GetMessage call, Windows fills in the fields of this structure with information about the message. The fields are as follows:

- *hwnd*: The handle for the window that is to receive the message.
- *iMessage*: A numeric code identifying the type of message (for example, keyboard or mouse).
- *wParam*: A 16-bit value containing information specific to the message. *See* The Windows Messages below.
- *lParam*: A 32-bit value containing information specific to the message.
- *time*: The time, in milliseconds, that the message was placed in the queue. The time is a 32-bit value relative to the time at which the current Windows session began.
- *pt.x*: The horizontal coordinate of the mouse cursor at the time the event occurred.
- *pt.y*: The vertical coordinate of the mouse cursor at the time the event occurred.

GetMessage always returns a nonzero value except when it receives a quit message. The quit message causes the message loop to end. The program should then terminate and return control to Windows.

Within the message loop, the TranslateMessage function translates physical keystrokes into character-code messages. Windows places these translated messages into the program's message queue.

The DispatchMessage function essentially makes a call to the window function of the window specified by the hwnd field. This window function (WndProc in SAMPLE) is indicated in the lpfnWndProc field of the window class structure.

When DispatchMessage passes the message to the window function, Windows uses the first four fields of the message structure as parameters to the function. The window function can then process the message. In SAMPLE, for instance, the four fields passed to WndProc are *hwnd* (the handle to the window), *iMessage* (the numeric message identifier), *wParam*, and *lParam*. Although Windows does not pass the time and mouse-position information fields as parameters to the window function, this information is available through the Windows functions GetMessageTime and GetMessagePos.

A Windows program obtains only a few specific types of messages through its message queue. These are keyboard messages, mouse messages, timer messages, the paint message that tells the program it must re-create the client area of its window, and the quit message that tells the program it is being terminated.

In addition to the queued messages, however, a program's window function also receives many nonqueued messages. Windows sends these nonqueued messages by bypassing the message loop and calling the program's window function directly.

Many of these nonqueued messages are derived from queued messages. For example, when the user clicks the mouse on the menu bar, a mouse-click message is placed in the program's message queue. The GetMessage function retrieves the message and the DispatchMessage function sends it to the program's window function. However, because this mouse message affects a nonclient area of the window (an area outside the window's client area), the window function normally does not process it. Instead, the function passes the message back to Windows. In this example, the message tells Windows to invoke a pop-up menu. Windows calls up the menu and then sends the window function several nonqueued messages to inform the program of this action.

A Windows program is thus message driven. Once a program reaches the message loop, it acts only when the window function receives a message. And, although a program receives many messages that affect the window, the program usually processes only some of them, sending the rest to Windows for normal default processing.

The Windows messages

Windows can send a window function more than 100 different messages. The WINDOWS.H header file includes identifiers for all these messages so that C programmers do not have to remember the message numbers. Some of the more common messages and the meanings of the *wParam* and *lParam* parameters are discussed here:

WM_CREATE. Windows sends a window function this nonqueued message while processing the CreateWindow call. The *lParam* parameter is a pointer to a creation structure. A window function can perform some program initialization during the WM_CREATE message.

WM_MOVE. Windows sends a window function the nonqueued WM_MOVE message when the window has been moved to another part of the display. The *lParam* parameter gives the new coordinates of the window relative to the upper left corner of the screen.

WM_SIZE. This nonqueued message indicates that the size of the window has been changed. The new size is encoded in the *lParam* parameter. Programs often save this window size for later use.

WM_PAINT. This queued message indicates that a region in the window's client area needs repainting. (The message queue can contain only one WM_PAINT message.)

WM_COMMAND. This nonqueued message signals a program that a user has selected a menu item or has triggered a keyboard accelerator. Child-window controls also use WM_COMMAND to send messages to the parent window.

WM_KEYDOWN. The *wParam* parameter of this queued message is a virtual key code that identifies the key being pressed. The *lParam* parameter includes flags that indicate the previous key state and the number of keypresses the message represents.

WM_KEYUP. This queued message tells a window function that a key has been released. The *wParam* parameter is a virtual key code.

WM_CHAR. This queued message is generated from WM_KEYDOWN messages during the TranslateMessage call. The *wParam* parameter is the ASCII code of a keyboard key.

WM_MOUSEMOVE. Windows uses this queued message to tell a program about mouse movement. The *lParam* parameter contains the coordinates of the mouse relative to the upper left corner of the client area of the window. The *wParam* parameter contains flags that indicate whether any mouse buttons or the Shift or Ctrl keys are currently pressed.

WM_xBUTTONDOWN. This queued message tells a program that a button on the mouse was depressed while the mouse was in the window's client area. The *x* can be either L, R, or M for the left, right, or middle mouse button. The *wParam* and *lParam* parameters are the same as for WM_MOUSEMOVE.

WM_xBUTTONUP. This queued message tells a program that the user has released a mouse button.

WM_xBUTTONDBLCLK. When the user double-clicks a mouse button, Windows generates a WM_*x*BUTTONDOWN message for the first click and a queued WM_*x*BUTTONDBLCLK message for the second click.

WM_TIMER. When a Windows program sets a timer with the SetTimer function, Windows places a WM_TIMER message in the message queue at periodic intervals. The *wParam* parameter is a timer ID. (If the message queue already contains a WM_TIMER message, Windows does not add another one to the queue.)

WM_VSCROLL. A Windows program that includes a vertical scroll bar in its window receives nonqueued WM_VSCROLL messages indicating various types of scroll-bar manipulation.

WM_HSCROLL. This nonqueued message indicates a user is manipulating a horizontal scroll bar.

WM_CLOSE. Windows sends a window function this nonqueued message when the user has selected *Close* from the window's system menu. A program can query the user to determine whether any action, such as saving a file to disk, is needed before the program is terminated.

WM_QUERYENDSESSION. This nonqueued message indicates that the user is shutting down Windows by selecting *Close* from the MS-DOS Executive system menu. A program can request the user to verify that the program should be ended. If the window function returns a zero value from the message, Windows does not end the session.

WM_DESTROY. This nonqueued message is the last message a window function receives before the program ends. A window function can perform some last-minute cleanup while processing WM_DESTROY.

WM_QUIT. This is a queued message that never reaches the window function because it causes GetMessage to return a zero value that causes the program to exit the message loop.

Message processing

Programmers can choose to process some messages and ignore others in the window function. Messages that are ignored are generally passed on to the function DefWindowProc for default processing within Windows.

Because Windows eventually has access to messages that a window function does not process, it can send a program messages that might otherwise be regarded as pertaining to system functions—for example, mouse messages that occur in a nonclient area of the window, or system keyboard messages that affect the menu. Unless these messages are passed on to DefWindowProc, the menu and other system functions do not work properly.

A program can, however, trap some of these messages to override Windows' default processing. For example, when Windows needs to repaint the nonclient area of a window (the title bar, system-menu box, and scroll bars), it sends the window function a WM_NCPAINT

(nonclient paint) message. The window function normally passes this message to DefWindowProc, which then calls routines to update the nonclient areas of the window. The program can, however, choose to process the WM_NCPAINT message and paint the nonclient area itself. A program that does this can, for example, draw its own scroll bars.

The Windows messaging system also notifies a program of important events occurring outside its window. For example, if the MS-DOS Executive were simply to end the Windows session when the user selects the *Close* option from its system menu, then applications that were still running would not have a chance to save changed files to disk. Instead, when the user selects *Close* from the last instance of the MS-DOS Executive's system menu, the MS-DOS Executive sends a WM_QUERYENDSESSION message to each currently running application. If any application responds by returning a zero value, the MS-DOS Executive does not end the Windows session.

Before responding, an application can process the WM_QUERYENDSESSION message and display a message box asking the user if a file should be saved. The message box should include three buttons labeled *Yes*, *No*, and *Cancel*. If the user answers *Yes*, the program can save the file and then return a nonzero value to the WM_QUERYENDSESSION message. If the user answers *No*, the program can return a nonzero value without saving the file. But if the user answers *Cancel*, the program should return a zero value so that the Windows session will not be ended. If a program does not process the WM_QUERYENDSESSION message, DefWindowProc returns a nonzero value.

When a user selects *Close* from the system menu of a particular instance of an application, rather than from the MS-DOS Executive's menu, Windows sends the window function a WM_CLOSE message. If the program has an unsaved file loaded, it can query the user with a message box—possibly the same one displayed when WM_QUERYENDSESSION is processed. If the user responds *Yes* to the query, the program can save the file and then call DestroyWindow. If the user responds *No*, the program can call DestroyWindow without saving the file. If the user responds *Cancel*, the window function does not call DestroyWindow and the program will not be terminated. If a program does not process WM_CLOSE messages, DefWindowProc calls DestroyWindow.

Finally, a window function can send messages to other window functions, either within the same program or in other programs, with the Windows SendMessage function. This function returns control to the calling program after the message has been processed. A program can also place messages in a program's message queue with the PostMessage function. This function returns control immediately after posting the message.

For example, when a program makes changes to the WIN.INI file (a file containing Windows initialization information), it can notify all currently running instances of these changes by sending them a WM_WININICHANGE message:

```
SendMessage (-1, WM_WININICHANGE, 0, 0L) ;
```

The -1 parameter indicates that the message is to be sent to all window functions of all currently running instances. Windows calls the window functions with the WM_WININICHANGE message and then returns control to the program that sent the message.

SAMPLE's message processing

The SAMPLE program shown in Figure 17-11 processes only four messages: WM_COMMAND, WM_SIZE, WM_PAINT, and WM_DESTROY. All other messages are passed to DefWindowProc. As is typical with most Windows programs written in C, SAMPLE uses a switch and case construction for processing messages.

The WM_COMMAND message signals the program that the user has selected a new font from the menu. SAMPLE first obtains a handle to the menu and removes the checkmark from the previously selected font:

```
hMenu = GetMenu (hWnd) ;
CheckMenuItem (hMenu, nCurrentFont, MF_UNCHECKED) ;
```

The value of *wParam* in the WM_COMMAND message is the menu ID of the newly selected font. SAMPLE saves that value in a static variable (nCurrentFont) and then places a checkmark on the new menu choice:

```
nCurrentFont = wParam ;
CheckMenuItem (hMenu, nCurrentFont, MF_CHECKED) ;
```

Because the typeface has changed, SAMPLE must repaint its display. The program does not repaint it immediately, however. Instead, it calls the InvalidateRect function:

```
InvalidateRect (hWnd, NULL, TRUE) ;
```

This causes a WM_PAINT message to be placed in the program's message queue. The NULL parameter indicates that the entire client area should be repainted. The TRUE parameter indicates that the background should be erased.

The WM_SIZE message indicates that the size of SAMPLE's client area has changed. SAMPLE simply saves the new dimensions of the client area in two static variables:

```
xClient = LOWORD (lParam) ;
yClient = HIWORD (lParam) ;
```

The LOWORD and HIWORD macros are defined in WINDOWS.H.

Windows also places a WM_PAINT message in SAMPLE's message queue when the size of the client area has changed. As is the case with WM_COMMAND, the program does not have to repaint the client area immediately, because the WM_PAINT message is in the message queue.

SAMPLE can receive a WM_PAINT message for many reasons. The first WM_PAINT message it receives results from calling UpdateWindow in the WinMain function. Later, if the current font is changed from the menu, the program itself causes a WM_PAINT message to be placed in the message queue by calling InvalidateRect. Windows also sends a window function a WM_PAINT message whenever the user changes the size of the window or when part of the window previously covered by another window is uncovered.

Programs begin processing WM_PAINT messages by calling BeginPaint:

```
BeginPaint (hWnd, &ps) ;
```

The SAMPLE program then creates a font based on the current size of the client area and the current typeface selected from the menu:

```
hFont = CreateFont (yClient, xClient / 8,
                    0, 0, 400, 0, 0, 0, OEM_CHARSET,
                    OUT_STROKE_PRECIS, OUT_STROKE_PRECIS,
                    DRAFT_QUALITY, (BYTE) VARIABLE_PITCH |
                    cFamily [nCurrentFont - IDM_SCRIPT],
                    szFace  [nCurrentFont - IDM_SCRIPT]) ;
```

The font is selected into the device context (a data structure internal to Windows that describes the characteristics of the output device); the program also saves the original device-context font:

```
hFont = SelectObject (ps.hdc, hFont) ;
```

And the word *Windows* is displayed:

```
TextOut (ps.hdc, 0, 0, "Windows", 7) ;
```

The original font in the device context is then selected, and the font that was created is now deleted:

```
DeleteObject (SelectObject (ps.hdc, hFont)) ;
```

Finally, SAMPLE calls EndPaint to signal Windows that the client area is now updated and valid:

```
EndPaint (hWnd, &ps) ;
```

Although the processing of the WM_PAINT message in this program is simple, the method used is common to all Windows programs. The BeginPaint and EndPaint functions always occur in pairs, first to get information about the area that needs repainting and then to mark that area as valid.

SAMPLE will display this text even when the program is minimized to be displayed as an icon at the bottom of the screen. Although most Windows programs use a customized icon for this purpose, the window-class structure in SAMPLE indicates that the program's icon is NULL, meaning that the program is responsible for drawing its own icon. SAMPLE does not, however, make any special provisions for drawing the icon. To it, the icon is simply a small client area. As a result, SAMPLE displays the word *Windows* in its "icon," using a small font size.

Windows sends the window function the WM_DESTROY message as a result of the DestroyWindow function that DefWindowProc calls when processing a WM_CLOSE message. The standard processing involves placing a WM_QUIT message in the message queue:

```
PostQuitMessage (0) ;
```

When the GetMessage function retrieves WM_QUIT from the message queue, GetMessage returns 0. This terminates the message loop and the program.

For all other messages, SAMPLE calls DefWindowProc and exits the window function by returning the value from the call:

```
return DefWindowProc (hWnd, iMessage, wParam, lParam) ;
```

This allows Windows to perform default processing on the messages SAMPLE ignores.

Windows' multitasking

Most operating systems or operating environments that allow multitasking use what is called a preemptive scheduler. Generally, the procedure involves use of the computer's clock to switch rapidly between programs and allow each a small time slice. When switching between programs, the operating system must preserve the machine state.

Windows is different. It is a nonpreemptive multitasking environment. Although Windows allows several programs to run simultaneously, it never switches from one program to allow another to run. It switches between programs only when the currently running program calls the GetMessage function or the related PeekMessage and WaitMessage functions.

When a Windows program calls GetMessage and the program's message queue contains a message other than WM_PAINT or WM_TIMER, Windows returns control to the program with the next message. However, if the program's message queue contains only a WM_PAINT or WM_TIMER message and another program's queue contains a message other than WM_PAINT or WM_TIMER, Windows returns control to the other program, which is also waiting for its GetMessage call to return.

(Windows also switches between programs temporarily when a program uses SendMessage to send a message to a window function in another program, but control returns to the calling program after the window function has processed the message sent to it.)

To cooperate with Windows' nonpreemptive multitasking, programmers should try to perform message processing as quickly as possible. Programs can, for example, split a large amount of processing into several smaller pieces to allow other programs to run in the interval. During long processing a program can also periodically call PeekMessage to allow other programs to run.

Graphics Device Interface

Programs receive input through the Windows message system. For program output, Windows provides a device-independent interface to graphics output devices, such as the video display, printers, and plotters. This interface is called the Graphics Device Interface, or GDI.

The device context (DC)

When a Windows program needs to send output to the video screen, the printer, or another graphics output device, it must first obtain a handle to the device's device context, or DC. Windows provides a number of functions for obtaining this device-context handle:

BeginPaint. Used for obtaining a video device-context handle during processing of a WM_PAINT message. This device context applies only to the rectangular section of the client area that is invalid (needs repainting). This region is also a clipping region, meaning that a program cannot paint outside this rectangle. BeginPaint fills in the fields of a PAINTSTRUCT structure. This structure contains the coordinates of the invalid rectangle and a byte that indicates if the background of the invalid rectangle has been erased.

GetDC. Generally used for obtaining a video device-context handle during processing of messages other than WM_PAINT. The handle obtained with this function references only the client area of the window.

GetWindowDC. Used for obtaining a video device-context handle that encompasses the entire window, including the title bar, menu bar, and scroll bars. A Windows program can use this function if it is necessary to paint over areas of the window outside the client area.

CreateDC. Used for obtaining a device-context handle for the entire display or for a printer, a plotter, or other graphics output device.

CreateIC. Used for obtaining an information-context handle, which is similar to a device-context handle but can be used only for obtaining information about the output device, not for drawing.

CreateCompatibleDC. Used for obtaining a device-context handle to a memory device context compatible with a particular graphics output device. This function is generally used for transferring bitmaps to a graphics output device.

CreateMetaFile. Used for obtaining a metafile device-context handle. A metafile is a collection of GDI calls encoded in binary form.

The Windows program uses the device-context handle when calling GDI functions. In addition to drawing, the various GDI functions can change the attributes of the device context, select different drawing objects (such as pens and fonts) into the device context, and determine the characteristics of the device context.

Device-independent programming

Windows supports such a wide variety of video displays, printers, and plotters that programs cannot make assumptions about the size and resolution of the device. Furthermore, because the user can generally alter the size of a program's window, the program must be able to adjust its output appropriately. The SAMPLE program, for example, showed how the window function can use the WM_SIZE message to obtain the current size of a window to create a font that fits text within the window's client area.

Programs can also use other Windows functions to determine the physical characteristics of a device. For instance, a program can use the GetDeviceCaps function to obtain

information about the device context, including the resolution of the device, its physical dimensions, and its relative pixel height and width.

Then, too, the GetTextMetrics function returns information about the current font selected in the device context. In the default device context, this is the system font. Many Windows programs base the size of their display output on the size of a system-font character.

Device-context attributes

The device context includes attributes that define how the graphics output functions work on the device. When a program first obtains a handle to a device context, Windows sets these attributes to default values, but the program can change them. Some of these device-context attributes are as follows:

Pen. Windows uses the current pen for drawing lines. The default pen produces a solid black line 1 pixel wide. A program can change the pen color, change to a dotted or dashed line, or make the pen draw a solid line wider than 1 pixel.

Brush. Windows uses the current brush (sometimes called a pattern) for filling areas. A brush is an 8-pixel-by-8-pixel bitmap. The default brush is solid white. Programs can create colored brushes, hatched brushes, and customized brushes based on bitmaps.

Background color. Windows uses the background color to fill the spaces in and between characters when drawing text and to color the open areas in hatched brushstrokes and dotted or dashed pen lines. Windows uses the background color only if the background mode (another attribute of the display context) is opaque. If the background mode is transparent, Windows leaves the background unaltered. The default background color is white.

Text color. Windows uses this color for drawing text. The default is black.

Font. Windows uses the font to determine the shape of text characters. The default is called the system font, a fixed-pitch font that also appears in menus, caption bars, and dialog boxes.

Additional device-context attributes (such as mapping modes) are described in the following sections.

Mapping modes

Most GDI drawing functions in Windows have parameters that specify the coordinates or size of an object. For instance, the Rectangle function has five parameters:

```
Rectangle (hDC, x1, y1, x2, y2) ;
```

The first parameter is the handle to the device context. The others are

- *x1*: horizontal coordinate of the upper left corner of the rectangle.
- *y1*: vertical coordinate of the upper left corner of the rectangle.
- *x2*: horizontal coordinate of the lower right corner of the rectangle.
- *y2*: vertical coordinate of the lower right corner of the rectangle.

In the Rectangle and most other GDI functions, coordinates are logical coordinates, which are not necessarily the same as the physical coordinates (pixels) of the device. To translate logical coordinates into physical coordinates, Windows uses the current mapping mode.

In actuality, the mapping mode defines a transformation of coordinates between a window, which is defined in terms of logical coordinates, and a viewport, which is defined in terms of physical coordinates. For any mapping mode, a program can define separate window and viewport origins. The logical point defined as the window origin is then mapped to the physical point defined as the viewport origin. For some mapping modes, a program can also define window and viewport extents, which determine how the logical coordinates are scaled to the physical coordinates.

Windows programs can select one of eight mapping modes. The first six are sometimes called fully constrained, because the ratio between the window and viewport extents is fixed and cannot be changed.

In MM_TEXT, the default mapping mode, coordinates on the x axis increase from left to right, and coordinates on the y axis increase from the top downward. In the other five fully constrained mapping modes, coordinates on the x axis also increase from left to right, but coordinates on the y axis increase from the bottom upward. The six fully constrained mapping modes are

- *MM_TEXT*: Logical coordinates are the same as physical coordinates.
- *MM_LOMETRIC*: Logical coordinates are in units of 0.1 millimeter.
- *MM_HIMETRIC*: Logical coordinates are in units of 0.01 millimeter.
- *MM_LOENGLISH*: Logical coordinates are in units of 0.01 inch.
- *MM_HIENGLISH*: Logical coordinates are in units of 0.001 inch.
- *MM_TWIPS*: Logical coordinates are in units of $\frac{1}{1440}$ inch. (These units are $\frac{1}{20}$ of a typographic point, which is approximately $\frac{1}{72}$ inch.)

The seventh mapping mode is called partially constrained, because a program can change the window and viewport extents but Windows adjusts the values to ensure that equal horizontal and vertical logical coordinates translate to equal horizontal and vertical physical dimensions:

- *MM_ISOTROPIC*: Logical coordinates represent the same physical distance on both the x and y axes.

The MM_ISOTROPIC mapping mode is useful for drawing circles and squares. The MM_LOMETRIC, MM_HIMETRIC, MM_LOENGLISH, MM_HIENGLISH, and MM_TWIPS mapping modes are also isotropic, because equal logical coordinates map to the same physical dimensions on both axes.

The final mapping mode is sometimes called unconstrained because a program is free to set different window and viewport extents on the x and y axes.

- *MM_ANISOTROPIC*: Logical coordinates are mapped to arbitrarily scaled physical coordinates.

Functions for drawing

Windows includes several functions that programs can use to draw in the client area of a window. The most common of these functions are

SetPixel. Sets a point to a particular color.

LineTo. Draws a line from the current position to a point specified in the LineTo function. The current position is defined in the device context and can be altered before the call to LineTo with the MoveTo function, which changes the current position but does not draw anything. Windows uses the current pen and the current drawing mode (*see* below) for drawing the line.

Polyline. Draws multiple lines much like a series of LineTo calls but does not alter the current position on completion.

Rectangle. Draws a filled rectangle with a border. Parameters to the Rectangle function specify the coordinates of the upper left and lower right corners of the rectangle. Windows draws the border of the rectangle with the current pen and current drawing mode defined in the device context, just as if it were using the Polyline function then Windows fills the rectangle with the current brush defined in the device context.

Ellipse. Uses the same parameters as Rectangle but draws an ellipse within the rectangular area.

RoundRect. Draws a rectangle with rounded corners. Two parameters to this function define the height and width of an ellipse that Windows uses for drawing the rounded corners.

Polygon. Draws a polygon connecting a series of points and fills the enclosed areas in either an alternate or winding mode. The winding mode causes Windows to fill every area within the polygon. The alternate mode fills every other area. For a polygon that defines a five-pointed star, for instance, the center is filled if the mode is winding but is not filled if the mode is alternate.

Arc. Draws a curved line that is part of the circumference of an ellipse.

Chord. Similar to the Arc function, but Windows connects the beginning and ending points of the arc with a straight line. The area is filled with the current brush defined in the device context.

Pie. Similar to the Arc function, but Windows draws lines from the beginning and ending points of the arc to the center of the ellipse. The area is filled with the current brush defined in the device context.

TextOut. Writes text with the current font, text color, background color, and background mode (transparent or opaque).

Windows also includes other drawing functions for filling areas, formatting text, and transferring bitmaps.

Raster operations for pens

When Windows uses a pen to write to a device context, it must first determine which pixels of the destination are to be altered by the pen (the foreground) and which pixels will not be affected (the background). With dotted and dashed pens, the background—the space between the dots or dashes—is left unaltered if the drawing mode is transparent and is filled with the background color if the drawing mode is opaque.

When Windows alters the pixels of the destination that correspond to the foreground of the pen, the most obvious result is that the color of the current pen defined in the display context is used to color the destination. But this is not the only possible result. Windows also generalizes the process by using a logical operation to combine the pixels of the pen and the pixels of the destination.

This logical operation is defined by the drawing mode attribute of the device context. This drawing mode can be set to one of 16 binary raster operations (abbreviated ROP2).

The following table shows the 16 binary raster operation codes defined in WINDOWS.H. The column headed "Resultant Destination" shows how the destination changes, depending on the bit pattern of the pen and the bit pattern of the destination before the line is drawn. The words OR, AND, XOR, and NOT are the logical operations.

Binary Raster Operation	Resultant Destination
R2_BLACK	0
R2_COPYPEN	pen
R2_MERGEPEN	pen OR destination
R2_MASKPEN	pen AND destination
R2_XORPEN	pen XOR destination
R2_NOTCOPYPEN	NOT pen
R2_NOTMERGEPEN	NOT (pen OR destination)
R2_NOTMASKPEN	NOT (pen AND destination)
R2_NOTXORPEN	NOT (pen XOR destination)
R2_MERGEPENNOT	pen OR (NOT destination)
R2_MASKPENNOT	pen AND (NOT destination)
R2_MERGENOTPEN	(NOT pen) OR destination
R2_MASKNOTPEN	(NOT pen) AND destination
R2_NOP	destination
R2_NOT	NOT destination
R2_WHITE	1

The default drawing mode defined in a device context is R2_COPYPEN, which simply copies the pen to the destination. However, if the pen color is blue, the destination is red, and the drawing mode is R2_MERGEPEN, then the drawn line appears as magenta, which

results from combining the pen and destination colors. If the pen color is blue, the destination is red, and the drawing mode is R2_NOTMERGEPEN, then the drawn line is green, because the blue pen and the red destination are combined into magenta, which Windows then inverts to make green.

Bit-block transfers

Windows also uses logical operations when transferring a rectangular pixel pattern (a bit block) from one device context to another or from one area of a device context to another area of the same device context.

While line drawing involves a logical combination of two sets of pixels (the pen and the destination), the bit-block transfer functions perform a logical combination of three sets of pixels: a source bitmap, a destination bitmap, and the brush currently selected in the destination device context. As shown in the preceding section, there are 16 different ROP2 drawing modes for all the possible combinations of two sets of pixels. The tertiary raster operations (abbreviated ROP3) for bit-block transfers require 256 different operations for all possible combinations.

Windows defines three functions for transferring rectangular pixel patterns: BitBlt (bit-block transfer), StretchBlt (stretch-block transfer), and PatBlt (pattern-block transfer). Of these three functions, StretchBlt is the most generalized. StretchBlt transfers a bitmap from a source device context to a destination device context. Function parameters specify the origin, width, and height of the bitmap. If the source and destination widths and heights are different, Windows stretches or compresses the bitmap appropriately. Negative values of widths and heights cause Windows to draw a mirror image of the bitmap.

The BitBlt function transfers a bitmap from a source device context to a destination device context, but the width and height of the source and destination must be the same. If the source and destination device contexts have different mapping modes, Windows uses StretchBlt instead.

In both BitBlt and StretchBlt, Windows performs a bit-by-bit logical operation with the bit block in the source device context, the bit block in the destination area of the destination device context, and the brush currently selected in the destination device context. Although Windows supports all 256 possible raster operations with these three bitmaps, only a few have been given WINDOWS.H identifiers:

Raster Operation	Resultant Destination
BLACKNESS	0
SRCCOPY	source
SRCAND	source AND destination
SRCPAINT	source OR destination

(more)

Raster Operation	Resultant Destination
SRCINVERT	source XOR destination
SRCERASE	source AND (NOT destination)
MERGEPAINT	source OR (NOT destination)
NOTSRCCOPY	NOT source
NOTSRCERASE	NOT (source OR destination)
DSTINVERT	NOT destination
PATCOPY	pattern
MERGECOPY	source AND pattern
PATINVERT	destination XOR pattern
PATPAINT	source OR (NOT destination) OR pattern
WHITENESS	1

The PatBlt function is similar to BitBlt and StretchBlt but performs a logical operation only between the currently selected brush and a destination area of the device context. Thus, only 16 raster operations can be used with PatBlt; these are equivalent to the binary raster operations used with line drawing.

Text and fonts

Windows supports file-based text fonts in two different formats: raster and vector. The raster fonts, such as Courier, Helvetica, and Times Roman, are defined by digital representations of the bit patterns of the characters. Font files usually contain several different sizes for each typeface. The vector fonts, such as Modern, Script, and Roman, are defined by points that are connected to form the letters and can be scaled to different sizes.

When using a device such as a printer, which has built-in fonts, Windows can also use these device-based fonts.

To specify a font, a Windows program uses the CreateFont function to create a logical font—a detailed description of the desired font. When this logical font is selected into a device context, Windows finds the actual font that best fits this description. In many cases, this match is not exact. The program can then call GetTextMetrics to determine the characteristics of the actual font that the device will use to display text.

Windows supports both fixed-width and variable-width fonts, as well as such attributes as italics, underlining, and boldfacing. Programs can also justify text with the GetTextExtent call, which obtains the width of a particular text string. The program can then insert extra spaces between words with SetTextJustification or it can insert extra spaces between letters with SetTextCharacterExtra.

Metafiles

As explained earlier, a metafile is a collection of GDI function calls stored in a binary coded form. A program can create a metafile by calling CreateMetaFile and giving it either

an MS-DOS filename or NULL as a parameter. If CreateMetaFile is given an MS-DOS filename, Windows creates a disk-based metafile; if the parameter is NULL, Windows creates a metafile in memory. The CreateMetaFile call returns a handle to a metafile device context. Any GDI calls that reference this device-context handle become part of the metafile.

When the program calls CloseMetaFile, Windows closes the metafile device context and returns a handle to the metafile. The program can then "play" this metafile on another device context (such as the video display) without calling the GDI functions directly.

Metafiles provide a useful way to transfer device-independent pictures between programs.

Data Sharing and Data Exchange

Windows includes a variety of methods by which programs can share and exchange data. These methods are discussed in the following sections.

Sharing local data among instances

Multiple instances of the same program can share data in the static data area of the program's data segment. Later instances of a program can thus call GetInstanceData and copy configuration options established by the user in the first instance. Multiple instances of programs can also share resources, such as dialog-box templates.

The Windows Clipboard

The Windows Clipboard is a general-purpose mechanism that allows a user to transfer data from one program to another. Programs that support the Clipboard generally include a top-level menu item called *Edit,* which invokes a pop-up menu that offers at least these three options:

- *Cut*: Copies the current selection to the Clipboard and deletes the selection from the current program file.
- *Copy*: Copies the current selection to the Clipboard without deleting the selection from the current program file.
- *Paste*: Copies the contents of the Clipboard to the current program file.

The Clipboard can hold only one item at a time. A program can transfer data to the Clipboard through the function call SetClipboardData. With this function, the program passes the Clipboard a handle to a global memory block, which then becomes the property of the Clipboard. A program can access Clipboard data through the complementary function GetClipboardData.

The Clipboard supports several formats:

- Text: ASCII text; each line ends with a carriage return and linefeed, and the text is terminated with a NULL character.
- Bitmap: A collection of bits in the GDI bitmap format.

- Metafile Picture: A structure that contains a handle to a metafile along with other information suggesting the mapping mode and aspect ratio of the picture.
- SYLK: Microsoft's Symbolic Link format.
- DIF: Software Arts' Data Interchange Format.

Programs can also use the Clipboard for storing data in private formats.

Some programs, such as the CLIPBRD program included with Windows, can also become Clipboard viewers. Such programs receive a message whenever the contents of the Clipboard change.

Dynamic Data Exchange (DDE)

Dynamic Data Exchange (DDE) is a protocol that cooperating programs can use to exchange data without user intervention. DDE makes use of the facilities in Windows that enable programs to send messages among themselves.

In DDE, the program that needs data from another program is called the client. The client sends a WM_DDE_INITIATE message either to a dedicated server program or to all currently running programs. Parameters to the WM_DDE_INITIATE message are *atoms*, which are numbers referring to text strings. A server application that has the data the client needs sends a WM_DDE_ACK message back to the client. The client can then be more specific about the data it needs by sending the server a WM_DDE_ADVISE message. The server can then pass global memory handles to the client with the WM_DDE_DATA message.

Internationalization

Windows includes several features that ease the conversion and translation of programs for international markets. Among these features are keyboard drivers appropriate for many European languages and use of the ANSI character set, which provides a richer set of accented letters than does the character set resident in the IBM PC and compatibles.

Windows also includes several functions that assist in language-independent coding. The AnsiUpper and AnsiLower functions translate characters or strings to uppercase or lowercase in the full ANSI character set, rather than the more limited ASCII character set. In addition, the AnsiNext and AnsiPrev functions allow scanning of text strings that may contain 2 or more bytes per character.

Windows programmers can also help in program translation by defining all text strings used within the program as resources contained in the resource script file. Because the resource script file also contains menu templates and dialog-box templates, it thus becomes the only file that needs alteration when a foreign-language version of the program is created.

Charles Petzold

Part E
Programming Tools

Article 18
Debugging in the MS-DOS Environment

It is axiomatic that any program will need debugging at some time in its development cycle, and programs written to run under MS-DOS are no exception. This article provides an introduction to the debugging tools and techniques available to the serious programmer developing code in the MS-DOS environment. Space does not permit a thorough investigation of the philosophy, psychology, and science of debugging computer programs; instead, a brief and practical discussion of the basic debugging approaches is presented, along with some rules-of-thumb for choosing the best approach. Nor are the details of every single utility and command included in this article; these are described in detail in the reference sections of this volume. The commands and utility programs that are most useful for debugging are discussed and illustrated with examples and case histories that also serve as models for the various debugging methods.

The reader of this article is assumed to be a programmer with sufficient experience to understand an assembly-language program. The reader is also assumed to be familiar with MS-DOS — terms like FCB and PSP are not explained. A reader without this background in MS-DOS need not be deterred, however; these terms are thoroughly explained elsewhere in this book. Besides assembly language, examples in this article are written in Microsoft QuickBASIC and Microsoft C. A detailed knowledge of these languages is not required; the examples are short and straightforward.

The reader should also keep in mind that the examples given here are real but not necessarily realistic. To avoid the tedium that accompanies debugging, the examples have been designed to reveal their bugs fairly quickly. All the methods and techniques shown are accurate in detail but not always in scale. Most of the debugging examples presented here would require one-half to one hour of work. It is possible for real debugging sessions to last for hours or days, especially if the wrong approach or tool is chosen. One of the purposes of this article is to help the programmer choose the correct tool and, thus, to reduce the tedium.

The Programs

There are more than a dozen listings in this article. Some of them are correct and others contain errors for use in illustrating debugging techniques. Many of the programs serve as examples in multiple sections of the article. The following summary of the programs (Table 18-1) is given to avoid confusion and to provide a common location to consult for explanations of the programs.

Table 18-1. Summary of Example Programs.

Name:	EXP.BAS
Figure:	18-1
Status:	Incorrect—do not use.
Purpose:	Computes EXP(x) (the exponential of x) to a specified precision using an infinite series.
Compiling:	QB EXP; LINK EXP;
Parameters:	Prompts for value for x and a convergence criterion. Enter zero to quit.

Name:	EXP.BAS
Figure:	18-3
Status:	Correct version of Figure 18-1.
Purpose:	Computes EXP(x) (the exponential of x) to a specified precision using an infinite series.
Compiling:	QB EXP; LINK EXP;
Parameters:	Prompts for value for x and a convergence criterion. Enter zero to quit.

Name:	COMMSCOP.ASM
Figure:	18-4
Status:	Correct.
Purpose:	Monitors the activity on a specified COM port and places a copy of all transmitted and received data in a RAM buffer. Each entry in the buffer is tagged to indicate whether the byte was sent by or received by the application program under test. Control is provided to start, stop, and resume tracing by means of a control interrupt. When tracing is stopped and resumed, a marker is left in the buffer. COMMSCOP is a terminate-and-stay-resident (TSR) program.
Compiling:	MASM COMMSCOP; LINK COMMSCOP; EXE2BIN COMMSCOP.EXE COMMSCOP.COM DEL COMMSCOP.EXE
Parameters:	Installed by entering *COMMSCOP*; no parameters for installation. The TSR is controlled by passing parameter data in registers with an Interrupt 60H call. The registers can have the following values:

AH:	Command:
00H	STOP
01H	FLUSH AND START
02H	RESUME TRACE
03H	RETURN TRACE BUFFER ADDRESS

(more)

DX:	COM port:
00H	COM1
01H	COM2

Interrupt 60H returns the following in response to function 3:

CX	Buffer count in bytes
DX	Segment address of buffer
BX	Offset address of buffer

Name: COMMSCMD.C
Figure: 18-5
Status: Correct.
Purpose: Controls the COMMSCOP program by issuing Interrupt 60H calls. C version.
COMPILING: MSC COMMSCMD;
LINK COMMSCMD;
Parameters: Commands are issued by
COMMSCMD [[*cmd*][*port*]]
where: *cmd* is the command to be executed:
STOP Stop trace
START Flush buffer and start trace
RESUME Resume a stopped trace
port is the COM port (1 = COM1, 2 = COM2)
If *cmd* is omitted, STOP is assumed; if *port* is omitted, 1 is assumed.

Name: COMMSCMD.BAS
Figure: 18-6
Status: Correct.
Purpose: Controls the COMMSCOP program by issuing Interrupt 60H calls. QuickBASIC version.
Compiling: QB COMMSCMD;
LINK COMMSCMD USERLIB;
Parameters: Commands are issued by
COMMSCMD [[*cmd*][,*port*]]
where: *cmd* is the command to be executed:
STOP Stop trace
START Flush buffer and start trace
RESUME Resume a stopped trace
port is the COM port (1 = COM1, 2 = COM2)
If *cmd* is omitted, STOP is assumed; if *port* is omitted, 1 is assumed.

Name COMMDUMP.BAS
Figure: 18-7
Status: Correct.
Purpose: Produces a formatted dump of the communications trace buffer.

(more)

Compiling: QB COMMDUMP;
LINK COMMDUMP USERLIB;
Parameters: No parameters. When COMMDUMP is invoked, it formats and dumps the entire buffer.

Name: TESTCOMM.ASM
Figure: 18-9
Status: Incorrect—do not use.
Purpose: Provides test data for the COMMSCOP routine.
Compiling: MASM TESTCOMM;
LINK TESTCOMM;
Parameters: No parameters. TESTCOMM reads data from the keyboard and writes to COM1 and reads COM1 data and displays it on the screen. Ctrl-C cancels.

Name: TESTCOMM.ASM
Figure: 18-10
Status: Correct version of Figure 18-9.
Purpose: Provides test data for the COMMSCOP routine.
Compiling: MASM TESTCOMM;
LINK TESTCOMM;
Parameters: No parameters. TESTCOMM reads data from the keyboard and writes to COM1 and reads COM1 data and displays it on the screen. Ctrl-C cancels.

Name: BADSCOP.ASM
Figure: 18-11
Status: Incorrect version of Figure 18-4—do not use.
Purpose: Monitors the activity on a specified COM port and places a copy of all transmitted and received data in a RAM buffer. Each entry in the buffer is tagged to indicate whether the byte was sent by or received by the application program under test. Control is provided to start, stop, and resume tracing by means of a control interrupt. When tracing is stopped and resumed, a marker is left in the buffer. BADSCOP is a terminate-and-stay-resident (TSR) program.
Compiling: MASM BADSCOP;
LINK BADSCOP;
EXE2BIN BADSCOP.EXE BADSCOP.COM
DEL BADSCOP.EXE
Parameters: Installed by entering *BADSCOP*; no parameters for installation. The TSR is controlled by passing parameter data in registers with an Interrupt 60H call. The registers can have the following values:

AH:	Command:
00H	STOP
01H	FLUSH AND START

(more)

02H	RESUME TRACE
03H	RETURN TRACE BUFFER ADDRESS
DX:	COM port:
00H	COM1
01H	COM2

Interrupt 60H returns the following in response to function 3:

CX	Buffer count in bytes
DX	Segment address of buffer
BX	Offset address of buffer

Name:	UPPERCAS.C
Figure:	18-13
Status:	Incorrect — do not use.
Purpose:	Converts a fixed string to uppercase and prints it.
Compiling:	MSC /Zi UPPERCAS; LINK UPPERCAS /CO;
Parameters:	No parameters.

Name:	UPPERCAS.C
Figure:	18-14
Status:	Correct version of Figure 18-13.
Purpose:	Converts a fixed string to uppercase and prints it.
Compiling:	MSC /Zi UPPERCAS; LINK UPPERCAS /CO;
Parameters:	No parameters.

Name:	ASCTBL.C
Figure:	18-16
Status:	Incorrect — do not use.
Purpose:	Displays a table of all displayable characters.
Compiling:	MSC /Zi ASCTBL; LINK ASCTBL /CO;
Parameters:	No parameters.

Name:	ASCTBL.C
Figure:	18-17
Status:	Correct version of Figure 18-16.
Purpose:	Displays a table of all displayable characters.
Compiling:	MSC /Zi ASCTBL; LINK ASCTBL /CO;
Parameters:	No parameters.

Debugging Tools and Techniques

MS-DOS provides a wide variety of tools to aid in the debugging process. Some are intended specifically for debugging. For example, the DEBUG program is delivered with MS-DOS and provides basic debugging aid; the more sophisticated SYMDEB is supplied with MASM, Microsoft's macro assembler; CodeView, a debugger for high-order languages, is supplied with Microsoft C, Microsoft Pascal, and Microsoft FORTRAN. Others are general MS-DOS services and features that are also useful in the debugging process.

Debugging, like programming, has aspects of both an art and a craft. The craft—the mechanical details of using the tools—is discussed both here and elsewhere in this volume, but the main subject of this article is the *art* of debugging—the choice of the correct tool, the best techniques to use in various situations, the methods of extracting the clues to the problem from a recalcitrant program.

Debugging a program is a form of puzzle solving. As with most intellectual detective work, the following rule applies:

Gather enough information and the solution will be obvious.

The craft of debugging involves gathering the data; the art lies in deciding which data to gather and in noticing when the solution has become obvious.

The methods of gathering data for debugging, listed in order of increasing difficulty and tediousness, fall into four major categories:

- Inspection and observation
- Instrumentation
- Use of software debugging monitors (DEBUG, SYMDEB, and CodeView)
- Use of hardware debugging aids

As mentioned above, part of the art of debugging is knowing which method to use. This is one of the most difficult aspects of debugging—so difficult, in fact, that even programmers with years of experience make mistakes. Many programmers have spent hours single-stepping through a program with DEBUG only to discover that the cause of the problem would have been obvious if they had given the program's output even a cursory check. The only universal rule for choosing the correct debugging method is

Try them all, starting with the simplest.

Inspection and observation

Inspection and observation is the oldest and, usually, the best method of program debugging. It is also the basis for all the other methods. The first step with this method, as with the others, is to gather all the pertinent materials. Program listings, file layouts, report layouts, and program design materials (such as algorithm descriptions and flowcharts) are all extremely valuable in the debugging process.

Desk-checking

Before a programmer can determine what a program is doing wrong, he or she must know the correct operation of the program. There was a time, when computers were rare and expensive resources, that programmers were encouraged not to run their programs until the programs had been thoroughly desk-checked. The desk-checking process involves sitting down with a listing, a hand calculator, and some sample data. The programmer then "plays computer," executing each line of the program manually and writing down on paper the results of each program step. This process is extremely slow and tedious. When the desk-checking is completed, however, the programmer not only has found most of the bugs in the program but also has become intimately familiar with the execution of the program and the values of the program variables at each step.

The advent of inexpensive yet powerful personal computers, combined with the rising cost of programmer time, has made complete desk-checking nearly obsolete. It is now cheaper to run the program and let the computer find the errors. However, the usefulness of the desk-checking technique remains. Many programmers find it helpful to manually execute those sections of a program that they suspect are causing trouble. Even if they don't find errors in the code, the insight they gain into the workings of the code and the values of the variables at each step can be invaluable when applying other debugging techniques.

The inspection-and-observation methodology

The basic technique of the inspection-and-observation method is simple: After gathering all the required materials, run the program and observe. Observe very carefully; events that seem insignificant may be the very clues needed to discover where the program is going astray. As the program executes, note whether each section performs correctly. Does the program clear the screen when it should? Does it ask for input when it should? Does it produce the correct results? Observable events are the debugger's milestones in the execution of the program. If the program clears the screen but writes purple asterisks instead of requesting input, then the problem lies somewhere after the program issues the Clear Screen command but before it writes the input prompt on the screen. At this point, the program listing and design data become important. Inspect the listing and examine the area after the last successful milestone and before the missing milestone. Look for a logic error in the code that could explain the observed data.

If the program produces printed reports, they may also be useful. Watch the screen and listen to the printer. Clues can sometimes be found in the order in which things happen. The light on the disk drive can be another indication of activity. See how disk activity coordinates with screen and printer events. Try to identify each section of the program from these clues. Then use this information to localize the inspection of the listing to isolate the erroneous code.

The values of data given in reports and on the screen can also give clues to what's going wrong. Examining the data and reconstructing the values used to compute it sometimes leads to inferences about data problems. Perhaps a variable was misspelled in the code

or perhaps a data file is in the wrong format or has been corrupted. With this information, the bug can often be isolated. However, a very thorough knowledge of the program and its algorithms is required. *See* Desk-checking above.

MS-DOS provides four commands and filters that are useful in the collection and examination of data for debugging: TYPE, PRINT, FIND, and DEBUG. All these commands display the data in a file in some way. If the data is ASCII (displayable) characters, TYPE and PRINT can be used, with some help from FIND. Binary files can be examined and modified with the DEBUG utility. *See* USER COMMANDS: FIND; PRINT; TYPE; PROGRAMMING UTILITIES: DEBUG.

The TYPE command provides the simplest way to display ASCII data files. This method can be used to examine both input and output files. Checking the input files may uncover some bad (or unexpected) data that causes the program to malfunction; examining the output files will show whether calculations are being performed correctly and may help pinpoint the erroneous calculations if they are not.

The FIND utility is useful in locating specific data in a file. Using FIND is more accurate and definitely less tedious than examining the file manually using the TYPE command. The /N switch causes FIND to also display the relative line number of the matching line—information that is most useful in debugging.

Sometimes the data is too complex to be examined on the screen and printed copy is needed. The PRINT command will produce hard copy of an ASCII file as will the TYPE command if its output is redirected to the printer with the >PRN command-line parameter after the filename.

Not all data files contain pure ASCII data, and displaying such non-ASCII files requires a different approach. The TYPE command can be used, but nonprintable characters will produce garbage on the screen. This technique can still prove useful if the file has a large amount of ASCII data or if the records are regular and the ASCII information always appears at the same location, but no information can be gained about non-ASCII numeric data in such files. Note also that the entire file might not be displayed using TYPE because if TYPE encounters a byte containing 1AH (Control-Z), it assumes it has reached the end of the file and stops.

Clearly, a more useful tool for examining non-ASCII files would be a program that dumps the file in hexadecimal, with an appropriate translation of all displayable characters. Such programs exist in the public domain (through bulletin-board services, for instance) and, in any event, are not difficult to write. MS-DOS does not include a stand-alone file-dumping program among its standard commands and utilities, but the DEBUG program can be used, with minor inconvenience, to display files. This program is discussed in detail later in this article; for now, simply follow these instructions to use DEBUG as a file dumper. To load DEBUG and the program to be debugged, use the form

DEBUG [*drive*:][*path*]*filename.ext*

DEBUG will display a hyphen as a prompt. To see the contents of the file, enter *D* (the DEBUG Display Memory command) and press Enter. DEBUG will display the first 128 (80H) bytes of the file in hexadecimal and will also show any displayable characters.

To see the rest of the file, simply continue entering *D* until the desired area is found. Hard copy of the contents of the display can be made by using the PrtSc key (or Ctrl-PrtSc to print continuously). Note that the offset addresses for the bytes in the file begin at the value in the program's CS:IP registers, which can be viewed by using the Debug R (Display or Modify Registers) command. To obtain the true offsets, subtract CS:IP from the address shown.

The essence of the inspection-and-observation method is careful and thoughtful observation. The computer and the operating system can provide tools to aid in the collection of data, but the most important tool is the programmer's mind. By applying the logical skills they already possess to the observed data, programmers can usually avoid the more complex forms of debugging.

Instrumentation

Debugging by instrumentation is a traditional method that has been popular since programs were holes punched in cards. In general, this method consists of adding something to the program, either internally or externally, to report on the progress of program execution. Programmers call this added mechanism instrumentation because of its resemblance to the measuring instruments used in science and engineering. Instrumentation can be software, hardware, or a combination of both; it can be internal to the program or external to it. Internal instrumentation is always software, but external instrumentation may be either hardware or software.

Internal instrumentation

Internal instrumentation usually consists of display or print statements placed at strategic locations. Other signals to the user can be used if they are available. For instance, the system beeper can be sounded at key locations, perhaps in a coded sequence of beeps; if the device being debugged has lights that can be accessed by the program, these lights can be flashed at important locations in the program. Beeping and flashing do not, however, possess the information content usually required for debugging, so display or print statements are preferred.

The use of display or print statements to display key data and milestones on the screen or printer requires careful planning. First, apply the techniques of inspection and observation described in the previous section to determine the most probable points of failure. Then, if there is some doubt about what path execution is taking through the code, embed messages of the following types after key decision points:

BEGINNING SORT PHASE
ENDING PRINCIPAL CALCULATION
PROCESSING RECORD *XX*

A second way to use display or print statement instrumentation is to embed statements that display the data and interim values used for calculations. This technique can be extremely useful in finding problems related to the data being processed. Consider the QuickBASIC program in Figure 18-1 as an example. The program has no syntax errors and compiles cleanly, but it sometimes produces an incorrect answer.

```
'  EXP.BAS -- COMPUTE EXPONENTIAL WITH INFINITE SERIES
'
' ************************************************************************
' *                                                                      *
' *  EXP                                                                 *
' *                                                                      *
' *  This routine computes EXP(x) using the following infinite series:   *
' *                                                                      *
' *                         x    x^2   x^3   x^4   x^5                   *
' *          EXP(x) = 1 + --- + --- + ---- + --- + --- + ...             *
' *                       1!    2!    3!    4!    5!                     *
' *                                                                      *
' *                                                                      *
' *  The program requests a value for x and a value for the convergence  *
' *  criterion, C.  The program will continue evaluating the terms of    *
' *  the series until the difference between two terms is less than C.   *
' *                                                                      *
' *  The result of the calculation and the number of terms required to   *
' *  converge are printed.  The program will repeat until an x of 0 is   *
' *  entered.                                                            *
' *                                                                      *
' ************************************************************************

'
'  Initialize program variables
'
INITIALIZE:
        TERMS = 1
        FACT = 1
        LAST = 1.E35
        DELTA = 1.E34
        EX = 1
'
'  Input user data
'
        INPUT "Enter number:  "; X
        IF X = 0 THEN END
        INPUT "Enter convergence criterion (.0001 for 4 places):  "; C

'
'  Compute exponential until difference of last 2 terms is < C
'
        WHILE ABS(LAST - DELTA) >= C
            LAST = DELTA
            FACT = FACT * TERMS
            DELTA = X^TERMS / FACT
            EX = EX + DELTA
            TERMS = TERMS + 1
        WEND
```

Figure 18-1. A routine to compute exponentials.

(more)

```
'
'  Display answer and number of terms required to converge
'
        PRINT EX
        PRINT TERMS; "elements required to converge"
        PRINT

        GOTO INITIALIZE
```

Figure 18-1. Continued.

The purpose of the EXP.BAS program is to compute the exponential of a given number to a specified precision using an infinite series. The program computes the value of each term in the infinite series and adds it to a running total. To keep from executing forever, the program checks the difference between the last two elements computed and stops when this difference is less than the convergence criterion entered by the user.

When the program is run for several values, the following results are observed:

```
Enter number:  ? 1
Enter convergence criterion (.0001 for 4 places):  ? .0001
 2.718282
 10 elements required to converge

Enter number:  ? 1.5
Enter convergence criterion (.0001 for 4 places):  ? .0001
 4.481686
 11 elements required to converge

Enter number:  ? 2
Enter convergence criterion (.0001 for 4 places):  ? .0001
 5
 3 elements required to converge

Enter number:  ? 2.5
Enter convergence criterion (.0001 for 4 places):  ? .0001
 12.18249
 15 elements required to converge

Enter number:  ? 3
Enter convergence criterion (.0001 for 4 places):  ? .0001
 13
 4 elements required to converge

Enter number:  ? 0
```

Some of these numbers are incorrect. Table 18-2 shows the computed values and the correct values.

Table 18-2. The Computed Values Generated by EXP.BAS and the Expected Values.

x	Computed	Correct
1.0	2.718282	2.718282
1.5	4.481686	4.481689
2.0	5	7.389056
2.5	12.18249	12.18249
3.0	13	20.08554

Applying the methods from the first section of this article and observing the data quickly reveals a pattern. With the exception of 1, all whole numbers give incorrect results, but all numbers with fractions give results that are correct to the specified convergence criterion. Examination of the listing shows no obvious reason for this. The answer is there, but only an exceptionally intuitive numeric analyst would see it. Because no answer is obvious, the next step is to validate the only information available — that whole numbers produce errors and fractional ones do not. Repeating the first experiment with 2 and a number very close to 2 yields the following results:

```
Enter number:  ? 1.999
Enter convergence criterion (.0001 for 4 places):  ? .0001
 7.38167
 13 elements required to converge

Enter number:  ? 2
Enter convergence criterion (.0001 for 4 places):  ? .0001
 5
 3 elements required to converge

Enter number:  ? 0
```

The outcome is the same — repeating the experiment with a number as near to 2 as the convergence criterion permits (1.9999) produces the same result. The error is indeed caused by the fact that the number is an integer.

Because no intuitive way can be found to solve the mystery by inspection, the programmer must turn to the next method in the hierarchy, instrumentation. The problem has something to do with the calculation of the terms of the series. Therefore, the section of the program that performs this calculation should be instrumented by placing PRINT statements inside the WHILE loop (Figure 18-2) to display all the intermediate values of the calculation.

```
WHILE ABS(LAST - DELTA) >= C
      LAST = DELTA
      FACT = FACT * TERMS
      DELTA = X ^ TERMS / FACT
```

Figure 18-2. Instrumenting the WHILE loop. *(more)*

```
            EX = EX + DELTA
            PRINT "TERMS="; TERMS; "EX="; EX; "FACT="; FACT; "DELTA="; DELTA;
            PRINT "LAST="; LAST
            TERMS = TERMS + 1
        WEND
```

Figure 18-2. Continued.

The print statements used in this WHILE loop are typical of the type used for instrumentation. The program makes no attempt at fancy formatting. The print statements simply identify each value with its variable name, allowing easy correlation of the data and the code in the listing. Repeating the experiment with 1.999 and 2 yields

```
Enter number:  ? 1.999
Enter convergence criterion (.0001 for 4 places):  ? .0001
TERMS= 1 EX= 2.999 FACT= 1 DELTA= 1.999 LAST= 1E+34
TERMS= 2 EX= 4.997001 FACT= 2 DELTA= 1.998 LAST= 1.999
TERMS= 3 EX= 6.328335 FACT= 6 DELTA= 1.331334 LAST= 1.998
TERMS= 4 EX= 6.993669 FACT= 24 DELTA= .6653343 LAST= 1.331334
TERMS= 5 EX= 7.25967 FACT= 120 DELTA= .2660006 LAST= .6653343
TERMS= 6 EX= 7.348292 FACT= 720 DELTA= 8.862254E-02 LAST= .2660006
TERMS= 7 EX= 7.373601 FACT= 5040 DELTA= 2.530806E-02 LAST= 8.862254E-02
TERMS= 8 EX= 7.379924 FACT= 40320 DELTA= 6.323853E-03 LAST= 2.530806E-02
TERMS= 9 EX= 7.381329 FACT= 362880 DELTA= 1.404598E-03 LAST= 6.323853E-03
TERMS= 10 EX= 7.38161 FACT= 3628800 DELTA= 2.807791E-04 LAST= 1.404598E-03
TERMS= 11 EX= 7.381661 FACT= 3.99168E+07 DELTA= 5.102522E-05 LAST= 2.807791E-04
TERMS= 12 EX= 7.38167 FACT= 4.790016E+08 DELTA= 8.499951E-06 LAST= 5.102522E-05
 7.38167
 13 elements required to converge

Enter number:  ? 2
Enter convergence criterion (.0001 for 4 places):  ? .0001
TERMS= 1 EX= 3 FACT= 1 DELTA= 2 LAST= 1E+34
TERMS= 2 EX= 5 FACT= 2 DELTA= 2 LAST= 2
 5
 3 elements required to converge
```

Examination of the instrumentation printout for the two cases shows a drastically different pattern. The fractional number went through 13 iterations following the expected pattern; the whole number, however, quit on the third step. The loop is terminating prematurely. Why? Look at the values calculated for *DELTA* and *LAST* on the last complete step. They are the same, giving a difference of zero. Because this difference will always be less than the convergence criterion, the loop will always terminate early. A moment's reflection shows why. The numerator of the fraction for each term but the first in the infinite series is a power of the number entered; the denominator is a factorial, a product formed by multiplying successive integers. Because $n! = n*(n-1)!$, when an integer is raised to a power equal to itself and divided by the factorial of that integer the result will always be the same as the preceding term. That is what has happened here.

Now that the cause of the problem is found, it must be fixed. How can this problem be prevented? In this case, the problem is caused by a logic error. The programmer misread (or miswrote!) the algorithm and assumed that the criterion for termination was that the difference between the last two terms be less than the specified value. This is incorrect. Actually, the termination criterion should be that the difference between the forming EXP(x) and the last term be less than the criterion. To simplify, the last term itself must be less than the value specified. The correct program listing, including the new WHILE loop, is shown in Figure 18-3.

```
'  EXP.BAS -- COMPUTE EXPONENTIAL WITH INFINITE SERIES
'
' ************************************************************************
' *                                                                      *
' *  EXP                                                                 *
' *                                                                      *
' *  This routine computes EXP(x) using the following infinite series:   *
' *                                                                      *
' *                          x    x^2   x^3   x^4   x^5                  *
' *         EXP(x) = 1 + --- + --- + --- + --- + --- + ...               *
' *                          1!    2!    3!    4!    5!                  *
' *                                                                      *
' *                                                                      *
' *  The program requests a value for x and a value for the convergence  *
' *  criterion, C.  The program will continue evaluating the terms of    *
' *  the series until the amount added with a term is less than C.       *
' *                                                                      *
' *  The result of the calculation and the number of terms required to   *
' *  converge are printed.  The program will repeat until an x of 0 is   *
' *  entered.                                                            *
' *                                                                      *
' ************************************************************************

'
'  Initialize program variables
'
INITIALIZE:
        TERMS = 1
        FACT = 1
        DELTA = 1.E35
        EX = 1
'
'  Input user data
'
        INPUT "Enter number:  "; X
        IF X = 0 THEN END
        INPUT "Enter convergence criterion (.0001 for 4 places):  "; C

'
'  Compute exponential until difference of last 2 terms is < C
'
```

Figure 18-3. Corrected exponential calculation routine. *(more)*

```
        WHILE DELTA > C
            FACT = FACT * TERMS
            DELTA = X^TERMS / FACT
            EX = EX + DELTA
            TERMS = TERMS + 1
        WEND

'
'  Display answer and number of terms required to converge
'
        PRINT EX
        PRINT TERMS; "elements required to converge"
        PRINT

        GOTO INITIALIZE
```

Figure 18-3. Continued.

The program now produces the correct results within the limits of the specified accuracy:

```
Enter number:  ? 1.999
Enter convergence criterion (.0001 for 4 places):  ? .0001
 7.381661
 12 elements required to converge

Enter number:  ? 2
Enter convergence criterion (.0001 for 4 places):  ? .0001
 7.389047
 12 elements required to converge

Enter number:  ? 0
```

This example illustrates how easy it is to use internal instrumentation in high-order languages. Because these languages usually have simple formatted output commands, they require very little work to instrument. When these output commands are not available, however, more work may be required. For instance, if the program being debugged is in assembly language, it is possible that the code required to format and print internal data will be longer than the program being debugged. For this reason, internal instrumentation is rarely used on small and moderate assembly programs. However, large assembly programs and systems often already have print formatting routines built into them; in these cases, internal instrumentation may be as easy as with high-order languages.

External instrumentation

Sometimes it is difficult to use internal instrumentation with a program. If, for instance, the problem is timing related, adding print statements could cloud the problem or, worse yet, make it go away completely. This leaves the programmer in the frustrating position of having the problem only when the cause can't be seen and not having the problem when it can. A solution to this type of problem can sometimes be found by moving the instrumentation outside the program itself. There are two types of external instrumentation: hardware and software.

Hardware instrumentation consists of whatever logic analyzers, oscilloscopes, meters, lights, bells, or gongs are appropriate to the hardware and software under test. Hardware instrumentation is difficult to set up and tedious to use. It is, therefore, usually reserved for those problems directly associated with hardware. Such problems often arise when new software is being run on new hardware and no one is quite sure where the bugs are. Because most programmers reading this book are developing software on tried-and-true personal computer hardware and because most of those programmers are unlikely to have a logic analyzer costing several thousand dollars, we will skip over the use of hardware instrumentation for software debugging. If a logic analyzer must be used, the programmer should remember that the debugging philosophy and techniques discussed in this article can still be applied effectively.

MS-DOS provides a feature that is very useful in building external instrumentation software: the TSR, or terminate-and-stay-resident routine. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: CUSTOMIZING MS-DOS: Terminate-and-Stay-Resident Utilities. This feature of the operating system allows the programmer to build a monitoring routine that is, in essence, a part of the operating system and outside the application program. The TSR is loaded as a normal program, but instead of leaving the system when it is done, it remains intact in memory. The operating system provides no way to reexecute the program after it terminates, so most TSRs are interrupt driven.

Because TSRs exist outside the application program, they can be used to build external instrumentation devices. This independence allows them to perform monitoring functions without disturbing the logic flow of the application program. The only areas where interference is likely are those where the TSR and the program must use common resources. These conflicts typically involve timing but can also involve other resources, such as I/O devices, disk files, and MS-DOS resources, including environment space. Some of these problems are addressed in the next example.

The TSR type of external instrumentation software can prove useful in analyzing serial communications. Such an instrumentation program monitors the serial communication line and records all data. To detect protocol or timing problems, the program tags the recorded data so that transmitted data can be differentiated from received data. Hardware devices exist that plug into the serial port and perform both the monitoring and tagging function, but they are expensive and not always handy. Fortunately, this inexpensive piece of software instrumentation will serve in many cases.

Software interrupt calls are made with the INT instruction. Although their service routines must obey similar rules, these interrupts should not be confused with hardware interrupts caused by external hardware events. Software interrupts in MS-DOS are used by an application program to communicate with the operating system and, by extension in IBM systems, with the ROM BIOS. For example, on IBM PCs and compatibles, application programs can use software Interrupt 14H to communicate with the ROM BIOS serial port driver. The ROM BIOS routine, in turn, manages the hardware interrupts from the actual

serial device. Thus, Interrupt 14H does not communicate directly with the hardware. All the programs in this article deal with software interrupts to the ROM BIOS and MS-DOS.

A program to trace the serial data flow must have access to the serial data, so such a program must replace the vector for Interrupt 14H with one that points to itself. The routine can then record all the serial data and pass it along through the serial port. Because the goal is to minimize the effect of this monitoring on the timing of the data, the method used for recording the data should be fast. This requirement rules out writing to a disk file, because unexpected delays can be introduced (and because doing disk I/O from an interrupt service routine under MS-DOS is difficult, if not impossible). Printing the data on paper is clearly too slow, and data displayed on the screen is too ephemeral. Thus, about the only thing that can be done with the data is to write it to RAM. Luckily, memory has become cheap and most personal computers have plenty.

Writing a routine that monitors and records serial data is not enough, however. The data must still flow through the serial port to and from the external serial device. Thus, the monitor program can have only temporary custody of the data and must pass it on to the serial interrupt service routine in the ROM BIOS. This is accomplished by using MS-DOS function calls to extract the address of the serial interrupt handler before the new vector is installed in its place. The process of intercepting interrupts and then passing the data on is known as "daisy-chaining" interrupt handlers. So long as such intercepting programs are careful to maintain the data and conditions upon entrance for subsequent routines (that is, so long as routines are well behaved; *see* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS), several interrupt handlers can be daisy-chained together with no detriment to processing. (Woe be unto the person who breaks the daisy chain—the results are annoying at best and unpredictable at worst.)

The serial monitoring program, as described so far, correctly collects and stores serial data and then passes it on to the serial port. This may be intellectually satisfying, but it is not of much use in the real world. Some way must be provided to control the program—to start collection, to stop collection, to pause and resume collection. Also, once data is collected, a control function must be provided that returns the number of bytes collected and their starting location, so that the data can be examined.

From all this, it is clear that a serial communications monitoring instrument must

1. Replace the Interrupt 14H vector with one pointing to itself.
2. Save the address of the old interrupt handler.
3. Collect the serial data, tag it as transmitted or received, and store it in RAM.
4. Pass the data on, in a completely transparent manner, to the old interrupt handler.
5. Provide some way to control data collection.

A program that meets all these criteria is shown in Figure 18-4. The COMMSCOP program has three major parts:

Procedure	Purpose
COMMSCOPE	Monitoring and tagging
CONTROL	External control
VECTOR_INIT	Interrupt vector initialization

The *COMMSCOPE* procedure provides the new Interrupt 14H handler that intercepts the serial I/O interrupts. The *CONTROL* procedure provides the external control needed to make the system work. The *VECTOR_INIT* procedure gets the old interrupt handler address, installs *COMMSCOPE* as the new interrupt handler, and installs the interrupt handler for the control interrupt.

```
        TITLE   COMMSCOP -- COMMUNICATIONS TRACE UTILITY
; ************************************************************************
; *                                                                      *
; *  COMMSCOP --                                                         *
; *     THIS PROGRAM MONITORS THE ACTIVITY ON A SPECIFIED COMM PORT      *
; *     AND PLACES A COPY OF ALL COMM ACTIVITY IN A RAM BUFFER.  EACH    *
; *     ENTRY IN THE BUFFER IS TAGGED TO INDICATE WHETHER THE BYTE       *
; *     WAS SENT BY OR RECEIVED BY THE SYSTEM.                           *
; *                                                                      *
; *     COMMSCOP IS INSTALLED BY ENTERING                                *
; *                                                                      *
; *                     COMMSCOP                                         *
; *                                                                      *
; *     THIS WILL INSTALL COMMSCOP AND SET UP A 64K BUFFER TO BE USED    *
; *     FOR DATA LOGGING.  REMEMBER THAT 2 BYTES ARE REQUIRED FOR        *
; *     EACH COMM BYTE, SO THE BUFFER IS ONLY 32K EVENTS LONG, OR ABOUT  *
; *     30 SECONDS OF CONTINUOUS 9600 BAUD DATA.  IN THE REAL WORLD,     *
; *     ASYNC DATA IS RARELY CONTINUOUS, SO THE BUFFER WILL PROBABLY     *
; *.    HOLD MORE THAN 30 SECONDS WORTH OF DATA.                         *
; *                                                                      *
; *     WHEN INSTALLED, COMMSCOP INTERCEPTS ALL INT 14H CALLS.  IF THE   *
; *     PROGRAM HAS BEEN ACTIVATED AND THE INT IS EITHER SEND OR RE-     *
; *     CEIVE DATA, A COPY OF THE DATA BYTE, PROPERLY TAGGED, IS PLACED  *
; *     IN THE BUFFER.  IN ANY CASE, DATA IS PASSED ON TO THE REAL       *
; *     INT 14H HANDLER.                                                 *
; *                                                                      *
; *     COMMSCOP IS INVOKED BY ISSUING AN INT 60H CALL.  THE INT HAS     *
; *     THE FOLLOWING CALLING SEQUENCE:                                  *
; *                                                                      *
; *             AH -- COMMAND                                            *
; *                   0 -- STOP TRACING, PLACE STOP MARK IN BUFFER       *
; *                   1 -- FLUSH BUFFER AND START TRACE                  *
; *                   2 -- RESUME TRACE                                  *
; *                   3 -- RETURN COMM BUFFER ADDRESSES                  *
; *             DX -- COMM PORT (ONLY USED WITH AH = 1 or 2)             *
; *                   0 -- COM1                                          *
; *                   1 -- COM2                                          *
```

Figure 18-4. Communications trace utility. *(more)*

```
; *                                                                    *
; *     THE FOLLOWING DATA IS RETURNED IN RESPONSE TO AH = 3:          *
; *                                                                    *
; *             CX -- BUFFER COUNT IN BYTES                            *
; *             DX -- SEGMENT ADDRESS OF THE START OF THE BUFFER       *
; *             BX -- OFFSET ADDRESS OF THE START OF THE BUFFER        *
; *                                                                    *
; *     THE COMM BUFFER IS FILLED WITH 2-BYTE DATA ENTRIES OF THE      *
; *     FOLLOWING FORM:                                                *
; *                                                                    *
; *             BYTE 0 -- CONTROL                                      *
; *                  BIT 0 -- ON FOR RECEIVED DATA, OFF FOR TRANS.     *
; *                  BIT 7 -- STOP MARK -- INDICATES COLLECTION WAS    *
; *                               INTERRUPTED AND RESUMED.             *
; *             BYTE 1 -- 8-BIT DATA                                   *
; *                                                                    *
; ********************************************************************

CSEG    SEGMENT
        ASSUME  CS:CSEG,DS:CSEG
        ORG     100H                    ;TO MAKE A COMM FILE

INITIALIZE:
        JMP     VECTOR_INIT             ;JUMP TO THE INITIALIZATION
                                        ; ROUTINE WHICH, TO SAVE SPACE,
                                        ; IS IN THE COMM BUFFER

;
;  SYSTEM VARIABLES
;
OLD_COMM_INT    DD      ?               ;ADDRESS OF REAL COMM INT
COUNT           DW      0               ;BUFFER COUNT
COMMSCOPE_INT   EQU     60H             ;COMMSCOPE CONTROL INT
STATUS          DB      0               ;PROCESSING STATUS
                                        ; 0 -- OFF
                                        ; 1 -- ON
PORT            DB      0               ;COMM PORT BEING TRACED
BUFPNTR         DW      VECTOR_INIT     ;NEXT BUFFER LOCATION

        SUBTTL  DATA INTERRUPT HANDLER
PAGE
; ********************************************************************
; *                                                                  *
; *  COMMSCOPE                                                       *
; *     THIS PROCEDURE INTERCEPTS ALL INT 14H CALLS AND LOGS THE DATA*
; *     IF APPROPRIATE.                                              *
; *                                                                  *
; ********************************************************************
COMMSCOPE       PROC    NEAR

        TEST    CS:STATUS,1             ;ARE WE ON?
        JZ      OLD_JUMP                ; NO, SIMPLY JUMP TO OLD HANDLER
```

Figure 18-4. Continued.

(more)

```
        CMP     AH,00H                  ;SKIP SETUP CALLS
        JE      OLD_JUMP                ; .

        CMP     AH,03H                  ;SKIP STATUS REQUESTS
        JAE     OLD_JUMP                ; .

        CMP     AH,02H                  ;IS THIS A READ REQUEST?
        JE      GET_READ                ; YES, GO PROCESS

;
;  DATA WRITE REQUEST -- SAVE IF APPROPRIATE
;
        CMP     DL,CS:PORT              ;IS WRITE FOR PORT BEING TRACED?
        JNE     OLD_JUMP                ; NO, JUST PASS IT THROUGH

        PUSH    DS                      ;SAVE CALLER'S REGISTERS
        PUSH    BX                      ; .
        PUSH    CS                      ;SET UP DS FOR OUR PROGRAM
        POP     DS                      ; .
        MOV     BX,BUFPNTR              ;GET ADDR OF NEXT BUFFER LOC
        MOV     [BX],BYTE PTR 0         ;MARK AS TRANSMITTED BYTE
        MOV     [BX+1],AL               ;SAVE DATA IN BUFFER
        INC     COUNT                   ;INCREMENT BUFFER BYTE COUNT
        INC     COUNT                   ; .
        INC     BX                      ;POINT TO NEXT LOCATION
        INC     BX                      ; .
        MOV     BUFPNTR,BX              ;SAVE NEW POINTER
        JNZ     WRITE_DONE              ;ZERO MEANS BUFFER HAS WRAPPED

        MOV     STATUS,0                ;TURN COLLECTION OFF
WRITE_DONE:
        POP     BX                      ;RESTORE CALLER'S REGISTERS
        POP     DS                      ; .
        JMP     OLD_JUMP                ;PASS REQUEST ON TO BIOS ROUTINE
;
;  PROCESS A READ DATA REQUEST AND WRITE TO BUFFER IF APPROPRIATE
;
GET_READ:
        CMP     DL,CS:PORT              ;IS READ FOR PORT BEING TRACED?
        JNE     OLD_JUMP                ; NO, JUST PASS IT THROUGH

        PUSH    DS                      ;SAVE CALLER'S REGISTERS
        PUSH    BX                      ; .
        PUSH    CS                      ;SET UP DS FOR OUR PROGRAM
        POP     DS                      ; .

        PUSHF                           ;FAKE INT 14H CALL
        CLI                             ; .
        CALL    OLD_COMM_INT            ;PASS REQUEST ON TO BIOS
        TEST    AH,80H                  ;VALID READ?
        JNZ     READ_DONE               ; NO, SKIP BUFFER UPDATE
```

Figure 18-4. Continued. *(more)*

```
        MOV     BX,BUFPNTR              ;GET ADDR OF NEXT BUFFER LOC
        MOV     [BX],BYTE PTR 1         ;MARK AS RECEIVED BYTE
        MOV     [BX+1],AL               ;SAVE DATA IN BUFFER
        INC     COUNT                   ;INCREMENT BUFFER BYTE COUNT
        INC     COUNT                   ; .
        INC     BX                      ;POINT TO NEXT LOCATION
        INC     BX                      ; .
        MOV     BUFPNTR,BX              ;SAVE NEW POINTER
        JNZ     READ_DONE               ;ZERO MEANS BUFFER HAS WRAPPED

        MOV     STATUS,0                ;TURN COLLECTION OFF
READ_DONE:
        POP     BX                      ;RESTORE CALLER'S REGISTERS
        POP     DS                      ; .
        IRET

;
;  JUMP TO COMM BIOS ROUTINE
;
OLD_JUMP:
        JMP     CS:OLD_COMM_INT

COMMSCOPE ENDP

        SUBTTL  CONTROL INTERRUPT HANDLER
PAGE
; ************************************************************************
; *                                                                      *
; *  CONTROL                                                             *
; *     THIS ROUTINE PROCESSES CONTROL REQUESTS.                         *
; *                                                                      *
; ************************************************************************

CONTROL PROC    NEAR
        CMP     AH,00H                  ;STOP REQUEST?
        JNE     CNTL_START              ; NO, CHECK START
        PUSH    DS                      ;SAVE REGISTERS
        PUSH    BX                      ; .
        PUSH    CS                      ;SET DS FOR OUR ROUTINE
        POP     DS
        MOV     STATUS,0                ;TURN PROCESSING OFF
        MOV     BX,BUFPNTR              ;PLACE STOP MARK IN BUFFER
        MOV     [BX],BYTE PTR 80H       ; .
        MOV     [BX+1],BYTE PTR 0FFH    ; .
        INC     BX                      ;INCREMENT BUFFER POINTER
        INC     BX                      ; .
        MOV     BUFPNTR,BX              ; .
        INC     COUNT                   ;INCREMENT COUNT
        INC     COUNT                   ; .
        POP     BX                      ;RESTORE REGISTERS
        POP     DS                      ; .
        JMP     CONTROL_DONE
```

Figure 18-4. Continued.

(more)

```
CNTL_START:
        CMP     AH,01H                  ;START REQUEST?
        JNE     CNTL_RESUME             ; NO, CHECK RESUME
        MOV     CS:PORT,DL              ;SAVE PORT TO TRACE
        MOV     CS:BUFPNTR,OFFSET VECTOR_INIT ;RESET BUFFER TO START
        MOV     CS:COUNT,0              ;ZERO COUNT
        MOV     CS:STATUS,1             ;START LOGGING
        JMP     CONTROL_DONE

CNTL_RESUME:
        CMP     AH,02H                  ;RESUME REQUEST?
        JNE     CNTL_STATUS             ; NO, CHECK STATUS
        CMP     CS:BUFPNTR,0            ;END OF BUFFER CONDITION?
        JE      CONTROL_DONE            ; YES, DO NOTHING
        MOV     CS:PORT,DL              ;SAVE PORT TO TRACE
        MOV     CS:STATUS,1             ;START LOGGING
        JMP     CONTROL_DONE

CNTL_STATUS:
        CMP     AH,03H                  ;RETURN STATUS REQUEST?
        JNE     CONTROL_DONE            ; NO, ERROR -- DO NOTHING
        MOV     CX,CS:COUNT             ;RETURN COUNT
        PUSH    CS                      ;RETURN SEGMENT ADDR OF BUFFER
        POP     DX                      ; .
        MOV     BX,OFFSET VECTOR_INIT   ;RETURN OFFSET ADDR OF BUFFER

CONTROL_DONE:
        IRET

CONTROL ENDP


        SUBTTL      INITIALIZE INTERRUPT VECTORS
PAGE
; ************************************************************************
; *                                                                      *
; *  VECTOR_INIT                                                         *
; *     THIS PROCEDURE INITIALIZES THE INTERRUPT VECTORS AND THEN        *
; *     EXITS VIA THE MS-DOS TERMINATE-AND-STAY-RESIDENT FUNCTION.       *
; *     A BUFFER OF 64K IS RETAINED.  THE FIRST AVAILABLE BYTE           *
; *     IN THE BUFFER IS THE OFFSET OF VECTOR_INIT.                      *
; *                                                                      *
; ************************************************************************

        EVEN                            ;ASSURE BUFFER ON EVEN BOUNDARY
VECTOR_INIT     PROC     NEAR
;
;  GET ADDRESS OF COMM VECTOR (INT 14H)
;
        MOV     AH,35H
```

Figure 18-4. Continued. *(more)*

```
        MOV     AL,14H
        INT     21H
;
;  SAVE OLD COMM INT ADDRESS
;
        MOV     WORD PTR OLD_COMM_INT,BX
        MOV     AX,ES
        MOV     WORD PTR OLD_COMM_INT[2],AX
;
;  SET UP COMM INT TO POINT TO OUR ROUTINE
;
        MOV     DX,OFFSET COMMSCOPE
        MOV     AH,25H
        MOV     AL,14H
        INT     21H
;
;  INSTALL CONTROL ROUTINE INT
;
        MOV     DX,OFFSET CONTROL
        MOV     AH,25H
        MOV     AL,COMMSCOPE_INT
        INT     21H
;
;  SET LENGTH TO 64K, EXIT AND STAY RESIDENT
;
        MOV     AX,3100H          ;TERM AND STAY RES COMMAND
        MOV     DX,1000H          ;64K RESERVED
        INT     21H               ;DONE
VECTOR_INIT ENDP
CSEG    ENDS
        END     INITIALIZE
```

Figure 18-4. Continued.

The first executable statement of the program is a jump to the *VECTOR_INIT* procedure. The vector initialization code is needed only during installation; after initialization of the vectors, the code can be discarded. In this case, the area where this code resides will become the start of the trace buffer; therefore, it makes sense to put the initialization code at the end of the program where it can be overlaid by the trace buffer. The jump at the start of the program is required because the rules for making .COM files require that the entry point be the first instruction of the program.

The vector initialization routine uses Interrupt 21H Function 35H (Get Interrupt Vector) to get the address of the current Interrupt 14H service routine. The segment and offset address (returned in the ES:BX registers) is stored in the doubleword at *OLD_COMM_INT*. Interrupt 21H Function 25H (Set Interrupt Vector) is then used to vector all Interrupt 14H calls to *COMMSCOPE*. Another Function 25H call sets Interrupt 60H to vector to the *CONTROL* routine. This interrupt, which provides the means to control and interrogate the *COMMSCOPE* routine, was chosen because it is unused by MS-DOS and because some IBM technical materials list 60H through 66H as being available for user interrupts. (If, for some reason, Interrupt 60H is not available, simply change the equated symbol *COMMSCOPE_INT* to an available interrupt.)

When the vector initialization process is complete, the routine exits and stays resident by using Interrupt 21H Function 31H (Terminate and Stay Resident). As part of the termination process, the routine requests 1000H paragraphs, or 64 KB, of storage. A little over 500 bytes of this storage area is used for the code; the rest is available for trace data. If the serial port is running at 2400 baud, a solid stream of data will fill this buffer in about two minutes. However, a solid 32 KB block of data is unusual in asynchronous communications and, in reality, the buffer will usually contain many minutes worth of data. Note that the buffer-handling routines in *COMMSCOPE* require that the buffer be aligned on an even byte boundary, so *VECTOR_INIT* is preceded by the EVEN directive.

The interrupt service routine, *COMMSCOPE*, receives all Interrupt 14H calls. First *COMMSCOPE* checks its own status. If it has not been activated, it immediately passes control to the real service routine. If the tracer is active, *COMMSCOPE* examines the Interrupt 14H function in AH. Setup and status requests (AH = 0 and AH = 3) do not affect tracing, so they are passed on directly to the the real service routine. If the Interrupt 14H call is a write-data request (AH = 1), *COMMSCOPE* moves the byte marking the data as transmitted and the data byte itself to the current buffer location and increments both the byte count and the buffer pointer by 2. If the buffer pointer goes to zero, the buffer has wrapped; data collection is turned off and cannot be turned on again without clearing the trace buffer. Because the buffer, which starts at *VECTOR_INIT*, is always on an even byte boundary, there is no danger of the first byte of the data pair forcing a wrap. After the transmitted data is added to the buffer, *COMMSCOPE* passes control to the real service routine.

A read-data request (AH = 2) must be handled a little differently. In this case, the data to be collected is not yet available. In order to get it, *COMMSCOPE* must pass control to the real service routine and then intercept the results on the way back. The code at *GET_READ* fakes an interrupt to the service routine by pushing the flags onto the stack so that the service routine's IRET will pop them off again. *COMMSCOPE* then calls the service routine and, when it returns, retrieves the incoming serial data character from AL. If the incoming data byte is valid (bit 7 of AH is zero), the byte marking the data as received and the data byte itself are placed in the trace buffer, and both the byte count and the buffer pointer are incremented by 2. The buffer-wrap condition is detected and handled in the same manner as with transmitted data. Because the real service routine has already been called, *COMMSCOPE* exits as if it were the service routine by issuing an IRET.

The *CONTROL* procedure provides the mechanism for external control of the trace procedure. The routine is entered whenever an Interrupt 60H is executed. Commands are sent through the AH register and can cause the routine to STOP (AH = 0), START/FLUSH (AH = 1), RESUME (AH = 2), or RETURN STATUS (AH = 3). This routine also sets the communications port to be traced. The required information is provided in DX using the same format as the Interrupt 14H routine. The port information is used only with START and RESUME requests. The RETURN STATUS command returns data in registers: the byte count (CX), the segment address of the buffer (DX), and the offset of the first byte in the buffer (BX).

The COMMSCOP program is assembled using the Microsoft Macro Assembler (MASM), linked using the Microsoft Object Linker (LINK), and then converted to a .COM file using EXE2BIN (*see* PROGRAMMING UTILITIES):

```
C>MASM COMMSCOP;  <Enter>
C>LINK COMMSCOP;  <Enter>
C>EXE2BIN COMMSCOP.EXE COMMSCOP.COM  <Enter>
C>DEL COMMSCOP.EXE  <Enter>
```

The linker will display the message *Warning: no stack segment*; this message can be ignored because the rules for making a .COM file forbid a separate stack segment.

The program is installed by simply typing *COMMSCOP*. Tracing can then be started and stopped using Interrupt 60H. MS-DOS does not allow resident routines to be removed, so COMMSCOP will be in the system until the system is restarted. Also note that, because COMMSCOP is well behaved, nothing disastrous will happen if multiple copies of it are accidentally installed. As each new copy is installed, it chains to the previous copy. When Interrupt 14H is intercepted, the new routine dutifully passes the data on to the previous routine, which repeats the process until the real service routine is reached. The data is added to the trace buffer of each copy, giving multiple, redundant copies of the same data. Because Interrupt 60H is not chained, only the last copy's buffer can be accessed. Thus, the other copies simply waste 64 KB each.

Two techniques can be used to start or stop a trace. The first is to issue Interrupt 60H calls at strategic locations within the program being debugged. With assembly-language programs, this is easy. The appropriate registers are loaded and an INT 60H instruction is executed. Issuing this INT instruction is not much more difficult with higher-order Microsoft languages—both QuickBASIC and C provide a library routine called INT86 that allows registers to be loaded and INT instructions to be executed. (In QuickBASIC, the INT86 library routine is included in the File USERLIB.OBJ; in Microsoft C, it is included in the file DOS.H.) Embedded Interrupt 60H calls can be convenient because they limit tracing to those areas where processing is suspect. Because COMMSCOP marks the buffer each time the trace is stopped and resumed, the separate pieces of a trace are easy to differentiate.

The second technique is to write a simple routine to start or stop the trace outside the program being debugged. The example in Figure 18-5, COMMSCMD, is a Microsoft C program that can perform these functions using the INT86 library function to issue Interrupt 60H calls.

```
/*************************************************************************
*                                                                        *
*  COMMSCMD                                                              *
*                                                                        *
*  This routine controls the COMMSCOP program that has been in-         *
*  stalled as a resident routine.  The operation performed is de-       *
*  termined by the command line.  The COMMSCMD program is invoked       *
*  as follows:                                                           *
*                                                                        *
*                COMMSCMD [[cmd][ port]]                                 *
*                                                                        *
```

Figure 18-5. A serial-trace control routine written in C. *(more)*

```
*  where cmd is the command to be executed                              *
*            STOP   -- stop trace                                       *
*            START  -- flush trace buffer and start trace               *
*            RESUME -- resume a stopped trace                           *
*        port is the COMM port to be traced (1=COM1, 2=COM2, etc.)      *
*                                                                       *
*  If cmd is omitted, STOP is assumed.  If port is omitted, 1 is        *
*  assumed.                                                             *
*                                                                       *
************************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include <dos.h>
#define COMMCMD 0x60

main(argc, argv)
int argc;
char *argv[];
{
    int cmd, port, result;
    static char commands[3] [10] = {"STOPPED", "STARTED", "RESUMED"};
    union REGS inregs, outregs;

    cmd = 0;
    port = 0;

    if (argc > 1)
        {
        if (0 == stricmp(argv[1], "STOP"))
            cmd = 0;
        else if (0 == stricmp(argv[1], "START"))
            cmd = 1;
        else if (0 == stricmp(argv[1], "RESUME"))
            cmd = 2;
        }

    if (argc == 3)
        {
        port = atoi(argv[2]);
        if (port > 0)
            port = port - 1;
        }

    inregs.h.ah = cmd;
    inregs.x.dx = port;
    result = int86(COMMCMD, &inregs, &outregs);


    printf("\nCommunications tracing %s for port COM%1d:\n",
            commands[cmd], port + 1);
}
```

Figure 18-5. Continued.

COMMSCMD is passed arguments in the command line. The first argument is the command to be performed: STOP, START, or RESUME. If no command is specified, STOP is assumed. The second argument is the port number: 1 (for COM1) or 2 (for COM2). If no port number is specified, 1 is assumed.

The COMMSCMD program uses a simple IF filter to determine the function to be performed. The program tests the number of arguments in the command line to see if a port has been specified. If the argument count (*argc*) is 3 (one for the command name, one for the command, and one for the port number), the port number argument is retrieved and converted to an integer. The Interrupt 60H routine expects port numbers to be specified in the same manner as for Interrupt 14H, so the port number is decremented if it is not already zero. The AH register is loaded with the command (*cmd*), the DX register is loaded with the port number (*port*), and the INT86 library function is then used to execute an Interrupt 60H call. When the interrupt returns, COMMSCMD displays a message showing the function and port.

The same function can be performed by the QuickBASIC program in Figure 18-6.

```
' ************************************************************************
' *                                                                      *
' *  COMMSCMD                                                            *
' *                                                                      *
' *  This routine controls the COMMSCOP program that has been in-        *
' *  stalled as a resident routine.  The operation performed is de-      *
' *  termined by the command line.  The COMMSCMD program is invoked      *
' *  as follows:                                                         *
' *                                                                      *
' *             COMMSCMD [[cmd][,port]]                                  *
' *                                                                      *
' *  where cmd is the command to be executed                             *
' *             STOP   -- stop trace                                     *
' *             START  -- flush trace buffer and start trace             *
' *             RESUME -- resume a stopped trace                         *
' *        port is the COMM port to be traced (1=COM1, 2=COM2, etc.)     *
' *                                                                      *
' *  If cmd is omitted, STOP is assumed.  If port is omitted, 1 is       *
' *  assumed.                                                            *
' *                                                                      *
' ************************************************************************

        '
        '  Establish system constants and variables
        '
        DEFINT A-Z

        DIM INREG(7), OUTREG(7)         'Define register arrays
```

Figure 18-6. A QuickBASIC version of COMMSCMD. *(more)*

```
        RAX = 0                         'Establish values for 8086
        RBX = 1                         '  registers
        RCX = 2                         '  .
        RDX = 3                         '  .
        RBP = 4                         '  .
        RSI = 5                         '  .
        RDI = 6                         '  .
        RFL = 7                         '  .

        DIM TEXT$(2)

        TEXT$(0) = "STOPPED"
        TEXT$(1) = "STARTED"
        TEXT$(2) = "RESUMED"

        '
        '  Process command-line tail
        '
        C$ = COMMAND$                   'Get command-line data

        IF LEN(C$) = 0 THEN             'If no command line specified
            CMD = 0                     'Set CMD to STOP
            PORT = 0                    'Set PORT to COM1
            GOTO SENDCMD
        END IF

        COMMA = INSTR(C$, ", ")         'Extract operands
        IF COMMA = 0 THEN
            CMDTXT$ = C$
            PORT = 0
        ELSE
            CMDTXT$ = LEFT$(C$, COMMA - 1)
            PORT = VAL(MID$(C$, COMMA + 1)) - 1
        END IF

        IF PORT < 0 THEN PORT = 0

        IF CMDTXT$ = "STOP" THEN
            CMD = 0
        ELSEIF CMDTXT$ = "START" THEN
            CMD = 1
        ELSEIF CMDTXT$ = "RESUME" THEN
            CMD = 2
        ELSE
            CMD = 0
        END IF

        '
        '  Send command to COMMSCOP routine
        '
SENDCMD:
        INREG(RAX) = 256 * CMD
```

Figure 18-6. Continued.

(more)

```
INREG(RDX) = PORT
CALL INT86(&H60, VARPTR(INREG(0)), VARPTR(OUTREG(0)))
'
'  Notify user that action is complete
'
PRINT : PRINT
PRINT "Communications tracing "; TEXT$(CMD);
IF CMD <> 0 THEN
    PRINT " for port COM"; MID$(STR$(PORT + 1), 2); ":"
ELSE
    PRINT
END IF

END
```

Figure 18-6. Continued.

Both versions of COMMSCMD accept their commands from the command tail; both are invoked with a STOP, START, or RESUME command and a serial port number (1 or 2). If the operands are omitted, STOP and COM1 are assumed.

After data has been collected and safely placed in the trace buffer, it must be read before it can be useful. Interrupt 60H provides a function (AH = 3) that returns the buffer address and the number of bytes in the buffer. The QuickBASIC routine in Figure 18-7 uses this function to get the address of the data and then formats the data on the screen.

```
' ************************************************************************
' *                                                                      *
' *  COMMDUMP                                                            *
' *                                                                      *
' *  This routine dumps the contents of the COMMSCOP trace buffer to     *
' *  the screen in a formatted manner.  Received data is shown in        *
' *  reverse video.  Where possible, the ASCII character for the byte    *
' *  is shown; otherwise a dot is shown.  The value of the byte is       *
' *  displayed in hex below the character.  Points where tracing was     *
' *  stopped are shown by a solid bar.                                   *
' *                                                                      *
' ************************************************************************

'
'  Establish system constants and variables
'
DEFINT A-Z

DIM INREG(7), OUTREG(7)          'Define register arrays

RAX = 0                          'Establish values for 8086
RBX = 1                          ' registers
RCX = 2                          '  .
RDX = 3                          '  .
```

Figure 18-7. Formatted dump routine for serial-trace buffer. *(more)*

```
RBP = 4                                 '  .
RSI = 5                                 '  .
RDI = 6                                 '  .
RFL = 7                                 '  .
                                           /
'
'  Interrogate COMMSCOP to obtain addresses and count of data in
'  trace buffer
'
INREG(RAX) = &H0300                    'Request address data and count
CALL INT86(&H60, VARPTR(INREG(0)), VARPTR(OUTREG(0)))

NUM = OUTREG(RCX)                      'Number of bytes in buffer
BUFSEG = OUTREG(RDX)                   'Buffer segment address
BUFOFF = OUTREG(RBX)                   'Offset of buffer start

IF NUM = 0 THEN END

'
'  Set screen up and display control data
'
CLS
KEY OFF
LOCATE 25, 1
PRINT "NUM ="; NUM;"BUFSEG = "; HEX$(BUFSEG); " BUFOFF = ";
PRINT HEX$(BUFOFF);
LOCATE 4, 1
PRINT STRING$(80,"-")
DEF SEG = BUFSEG

'
'  Set up display control variables
'
DLINE = 1
DCOL = 1
DSHOWN = 0

'
'  Fetch and display each character in buffer
'
FOR I= BUFOFF TO BUFOFF+NUM-2 STEP 2
    STAT = PEEK(I)
    DAT = PEEK(I + 1)

    IF (STAT AND 1) = 0 THEN
        COLOR 7, 0
    ELSE
        COLOR 0, 7
    END IF

    RLINE = (DLINE-1) * 4 + 1
```

Figure 18-7. Continued. *(more)*

```
        IF (STAT AND &H80) = 0 THEN
            LOCATE RLINE, DCOL
            C$ = CHR$(DAT)
            IF DAT < 32 THEN C$ = "."
            PRINT C$;
            H$ = RIGHT$("00" + HEX$(DAT), 2)
            LOCATE RLINE + 1, DCOL
            PRINT LEFT$(H$, 1);
            LOCATE RLINE + 2, DCOL
            PRINT RIGHT$(H$, 1);
        ELSE
            LOCATE RLINE, DCOL
            PRINT CHR$(178);
            LOCATE RLINE + 1, DCOL
            PRINT CHR$(178);
            LOCATE RLINE + 2, DCOL
            PRINT CHR$(178);
        END IF

        DCOL = DCOL + 1
        IF DCOL > 80 THEN
            COLOR 7, 0
            DCOL = 1
            DLINE = DLINE + 1
            SHOWN = SHOWN + 1
            IF SHOWN = 6 THEN
                LOCATE 25, 50
                COLOR 0, 7
                PRINT "ENTER ANY KEY TO CONTINUE:  ";
                WHILE LEN(INKEY$) = 0
                WEND
                COLOR 7, 0
                LOCATE 25, 50
                PRINT SPACE$(29);
                SHOWN = 0
            END IF
            IF DLINE > 6 THEN
                LOCATE 24, 1
                PRINT : PRINT : PRINT : PRINT
                LOCATE 24, 1
                PRINT STRING$(80, "-");
                DLINE = 6
            ELSE
                LOCATE DLINE * 4, 1
                PRINT STRING$(80, "-");
            END IF
        END IF

    NEXT I

    END
```

Figure 18-7. Continued.

COMMDUMP is a simple routine. Like most debugging aids, it lacks needless frills. When it is executed, COMMDUMP displays the data in the trace buffer on the screen in the format shown in Figure 18-8.

```
.012832.132056780001006713205678000100671320567800010067132056780001006713205678
033333303333333333333333333333333333333333333333333333333333333333333333333333
101283231320567800010067132056780001006713205678000100671320567800010067132056 78
--------------------------------------------------------------------------------
000100671320567800010067132056780001006713205678000100 67.#.▌.012832.567813200001
333333333333333333333333333333333333333333333333333333333021▌033333303333333333
000100671320567800010067132056780001006713205678000100673383▌101283235678132000 01
--------------------------------------------------------------------------------
006756781320000100675678132000010067567813200001006756781320000100675678132000 01
333333333333333333333333333333333333333333333333333333333333333333333333333333
006756781320000100675678132000010067567813200001006756781320000100675678132000 01
--------------------------------------------------------------------------------
006756781320000100675678132000010067.#.▌.012832.006713205678000100671320567800 01
333333333333333333333333333333333333021▌033333303333333333333333333333333333333
006756781320000100675678132000010067338▌101283230067132056780001006713205678000 1
--------------------------------------------------------------------------------
006713205678000100671320567800010067132056780001006713205678000100671320567800 01
333333333333333333333333333333333333333333333333333333333333333333333333333333
006713205678000100671320567800010067132056780001006713205678000100671320567800 01
--------------------------------------------------------------------------------
006713205678000 1.#.▌.012832.132056780001006713205678000100671320567800010067132 0
333333333333333021▌033333303333333333333333333333333333333333333333333333333333
006713205678000 1338▌101283231320567800010067132056780001006713205678000100671320
--------------------------------------------------------------------------------
NUM = 1122 BUFSEG = 1313 BUFOFF = 208          ENTER ANY KEY TO CONTINUE:
```

Figure 18-8. Formatted trace dump routine output.

Note that the data for each byte is presented in two forms. If the byte is greater than 1FH, the ASCII character represented by that number is shown; otherwise, a dot is shown. Directly below each character is the hexadecimal representation of the data. The display shows received data in reverse video and transmitted data in normal video. The mark placed in the buffer when collection is stopped and resumed is represented on the screen as a vertical bar one character wide. The display pauses when the screen is full and waits for a key to be pressed.

Data collected and displayed in this way can be invaluable to the programmer trying to debug a program involving a communications protocol. The example shown above is part of an ordered exchange of sales data for a system using blocked transmissions and ACK/NAK protocol. Like all debugging, finding bugs in such a system requires the collection of large amounts of data. With no data, the causes of problems can be almost impossible to find; with sufficiently large amounts of data, the solutions are obvious.

Several things could be done to the COMMSCOP program to increase its usefulness. For instance, there are six unused bits in the tag accompanying each data byte in the trace buffer. These could be used to record the status of the modem control bits, to place timer ticks in the buffer, or to coordinate the data with some outside event. (Such changes to COMMSCOP would require a more complicated COMMDUMP routine to display them.)

Software debugging monitors

Debugging monitors provide the next level of sophistication in the hierarchy of debugging methods. These monitors are coresident in memory with the application being debugged and provide a controlled testing environment—that is, they allow the programmer to control the execution of the program and to monitor the results. They even allow some problems to be fixed directly and the result reexecuted immediately, without the need to reassemble or recompile.

These monitors are analogous to the TSR serial monitor from the previous section. The debugging monitors, however, do not reside permanently in memory and are controlled interactively from the keyboard during the execution of the program under test. Although this level of control is more flexible than instrumentation, it is also more intrusive into program execution. While the debugging monitor sits and waits for input from the keyboard, the application program is also idle. For programs that must run in real time or must respond to external stimuli, long delays can be fatal. Careful planning and a thorough knowledge of the internal workings of the program are required to debug in such an environment.

Other problems with debugging monitors arise from the nature of the monitors themselves. They are programs, no different from the application program being debugged and are therefore limited to those things that can be done with software. For instance, they can break (stop execution to allow investigation of program status) when a specific instruction address is executed (because this can be done with software), but they cannot break when a data address is referenced (because this would require special hardware). Because these monitors reside in RAM, as do the application program and MS-DOS, they are susceptible to damage from a program running wild. Some trial and error is usually involved in locating the problem causing this kind of damage; breakpoints won't work here because the problem kills the monitor (and usually MS-DOS also).

Microsoft provides three debugging monitors, each with greater capabilities than its predecessor. In order of increasing sophistication, these three monitors are

Monitor	Description
DEBUG	A basic debugging monitor with the ability to load files, modify memory and registers, execute programs, set simple breakpoints, trace execution, modify disk files, and enter assembly-language statements into memory.
SYMDEB	A more advanced debugging monitor incorporating all the features of DEBUG plus more sophisticated data display, support for graphics programs, support for the Intel 80186/80286 microprocessors and the Intel 80287 math coprocessor, improved breakpoints, improved tracing, recognition of symbols from the program being debugged, and limited source-line display.
CodeView	The most sophisticated debugging monitor, incorporating the functionality of SYMDEB (with some differences in the details) plus windows, full source-line support, mouse support, and generally more sophistication on all functions.

Although all these debugging monitors will be discussed here, this section is not intended to be a tutorial on all the commands and options of the monitors—those are presented elsewhere in this volume and in the manuals accompanying the monitors. *See* PROGRAMMING UTILITIES: DEBUG; SYMDEB; CODEVIEW. Rather, this section uses case histories and sample programs to illustrate the techniques for solving various types of common debugging problems. The case histories have been chosen to show a wide range of problems, from simple to extremely complex.

DEBUG

Although DEBUG is the least sophisticated of the software debugging monitors, it is quite useful with moderately complex programs and is an effective tool for learning basic techniques.

Basic techniques

The first sample program is written in assembly language. It is a test program that performs serial input and output and was used to debug COMMSCOP, the serial-trace TSR presented earlier. The routine reads from the keyboard and writes to COM1 by means of Interrupt 14H. It also accepts incoming serial data and displays it on the screen. This process continues until Ctrl-C is pressed on the keyboard. A serial terminal is attached to COM1 to serve as a data source. Figure 18-9 shows the erroneous program.

```
        TITLE TESTCOMM - TEST COMMSCOP ROUTINE
; ************************************************************************
; *                                                                      *
; *  TESTCOMM                                                            *
; *     THIS ROUTINE PROVIDES DATA FOR THE COMMSCOP ROUTINE.  IT READS   *
; *     CHARACTERS FROM THE KEYBOARD AND WRITES THEM TO COM1 USING       *
; *     INT 14H.  DATA IS ALSO READ FROM INT 14H AND DISPLAYED ON THE    *
; *     SCREEN.  THE ROUTINE RETURNS TO MS-DOS WHEN Ctrl-C IS PRESSED    *
; *     ON THE KEYBOARD.                                                 *
; *                                                                      *
; ************************************************************************

SSEG    SEGMENT PARA STACK 'STACK'
        DW      128 DUP(?)
SSEG    ENDS

CSEG    SEGMENT
        ASSUME  CS:CSEG,SS:SSEG
BEGIN   PROC    FAR
        PUSH    DS                      ;SET UP FOR RET TO MS-DOS
        XOR     AX,AX                   ; .
        PUSH    AX                      ; .
```

Figure 18-9. Incorrect serial test routine. *(more)*

```
MAINLOOP:
        MOV     AH,6                    ;USE MS-DOS CALL TO CHECK FOR
        MOV     DL,0FFH                 ; KEYBOARD ACTIVITY
        INT     21                      ; IF NO CHARACTER, JUMP TO
        JZ      TESTCOMM                ; COMM ACTIVITY TEST

        CMP     AL,03                   ;WAS CHARACTER A Ctrl-C?
        JNE     SENDCOMM                ; NO, SEND IT TO SERIAL PORT
        RET                             ; YES, RETURN TO MS-DOS

SENDCOMM:
        MOV     AH,01                   ;USE INT 14H WRITE FUNCTION TO
        MOV     DX,0                    ; SEND DATA TO SERIAL PORT
        INT     14H                     ; .

TESTCOMM:
        MOV     AH,3                    ;GET SERIAL PORT STATUS
        MOV     DX,0                    ; .
        INT     14H                     ; .
        AND     AH,1                    ;ANY DATA WAITING?
        JZ      MAINLOOP                ; NO, GO BACK TO KEYBOARD TEST
        MOV     AH,2                    ;READ SERIAL DATA
        MOV     DX,0                    ; .
        INT     14H                     ; .
        MOV     AH,6                    ;WRITE SERIAL DATA TO SCREEN
        INT     21H                     ; .
        JMP     MAINLOOP                ;CONTINUE
BEGIN   ENDP
CSEG    ENDS
        END     BEGIN
```

Figure 18-9. Continued.

When executed, this program produces a constant stream of zeros from the serial port. Incoming serial data is not echoed on the screen, but the cursor moves as if it were. Further, the Ctrl-C keystroke is not recognized, so the only way to stop the program is to restart the system.

An examination of the listing should reveal the errors that cause these problems, but things do not always happen that way. For the purposes of this case study, assume that the listing was no help. Instrumentation is more difficult for assembly-language programs than for programs written in higher-order languages, so in this case it is advantageous to go directly to a debugging monitor. The monitor for this example is DEBUG.

The first step in using DEBUG is not to invoke the monitor; rather, it is to gather all pertinent listings, link maps, and program design documentation. In this case, the program is so short that a link map will not be needed; all the design documentation that exists is in the program comments.

Now begin DEBUG by typing

```
C>DEBUG TESTCOMM.EXE  <Enter>
```

The filename must be fully qualified; DEBUG makes no assumptions about the extension. Any type of file can be examined with DEBUG, but only files with an extension of .COM, .EXE, or .HEX are actually loaded and made ready for execution. Since TESTCOMM is a .EXE file, DEBUG loads it and prepares it for execution in a manner compatible with the MS-DOS loader. Type the Display or Modify Registers command, R.

```
-R  <Enter>
AX=0000  BX=0000  CX=0131  DX=0000  SP=0100  BP=0000  SI=0000  DI=0000
DS=1AAD  ES=1AAD  SS=1ABD  CS=1ACD  IP=0000   NV UP EI PL NZ NA PO NC
1ACD:0000 1E            PUSH  DS
```

Notice that the SS and CS registers have been loaded to their correct values and that SP points to the bottom of the stack. DS and ES point to an address 100H bytes (10H paragraphs) before the stack segment. (This is because the system sets these registers to point to the program segment prefix [PSP] when a .EXE program is loaded.) Normally, the program code would be responsible for loading the correct value of DS, but this example does not use the data segment, so the program doesn't bother. The register display also shows the instruction at the current value of CS:IP, 1ACD:0000H. The instruction pointer was set to this address because the END statement in the source program specified the procedure *BEGIN* as the entry point and that procedure begins at CS:IP. Note that the instruction displayed below the register information has not yet been executed. This condition is true for all register displays in DEBUG — IP always points to the *next* instruction to be executed, so the instruction at IP has not been executed.

From the symptoms observed during program execution, it is clear that the keyboard data is not reaching the serial port. The failure could be in the keyboard read routine or in the serial port write routine. This code is compact and fairly linear, so the easiest way to find out what is going on is to trace through the first few instructions of the program. Executing five instructions with the Trace Program Execution command, T, will do this.

```
-T5  <Enter>

AX=0000  BX=0000  CX=0131  DX=0000  SP=00FE  BP=0000  SI=0000  DI=0000
DS=1AAD  ES=1AAD  SS=1ABD  CS=1ACD  IP=0001   NV UP EI PL NZ NA PO NC
1ACD:0001 33C0          XOR   AX,AX

AX=0000  BX=0000  CX=0131  DX=0000  SP=00FE  BP=0000  SI=0000  DI=0000
DS=1AAD  ES=1AAD  SS=1ABD  CS=1ACD  IP=0003   NV UP EI PL ZR NA PE NC
1ACD:0003 50            PUSH  AX

AX=0000  BX=0000  CX=0131  DX=0000  SP=00FC  BP=0000  SI=0000  DI=0000
DS=1AAD  ES=1AAD  SS=1ABD  CS=1ACD  IP=0004   NV UP EI PL ZR NA PE NC
1ACD:0004 B406          MOV   AH,06

AX=0600  BX=0000  CX=0131  DX=0000  SP=00FC  BP=0000  SI=0000  DI=0000
DS=1AAD  ES=1AAD  SS=1ABD  CS=1ACD  IP=0006   NV UP EI PL ZR NA PE NC
1ACD:0006 B2FF          MOV   DL,FF

AX=0600  BX=0000  CX=0131  DX=00FF  SP=00FC  BP=0000  SI=0000  DI=0000
DS=1AAD  ES=1AAD  SS=1ABD  CS=1ACD  IP=0008   NV UP EI PL ZR NA PE NC
1ACD:0008 CD15          INT   15
```

The Trace command shows the contents of the registers as each instruction is executed. The register contents are *after* the execution of the instruction listed above the registers and the instruction shown with the registers is the *next* instruction to be executed. The first register display in this example represents the state of affairs after the execution of the PUSH DS instruction, as indicated by SP. The first three instructions set up the stack so that the far return issued at the end of the program will pass control to the PSP for termination. The next two instructions set the registers for a Direct Console I/O MS-DOS call (AH = 060, DL = HFFH for input). After these registers are set up, the program should execute the MS-DOS call INT 21H. However, the next instruction to be executed is INT 15H. This is the reason the keyboard data is not being read. The code requests INT 21, not 21H. This mistake is a common one. The assembler's default radix is decimal, so it converted 21 into 15H. This error can be corrected in memory from within DEBUG and, because the instruction hasn't executed yet, the fix can be tested immediately. To make the correction, use the Assemble Machine Instructions command, A.

```
-A 8  <Enter>
1ACD:0008 int 21  <Enter>
1ACD:000A  <Enter>
```

The *A 8* code instructs DEBUG to begin assembling at CS:0008H. DEBUG prompts with the address and waits for an instruction to be entered. The letter *H* is not needed after the 21 this time because DEBUG assumes all numbers entered with the Assemble command are in hexadecimal form. In general, any valid 8086/8087/8088 assembly-language statement can be entered this way and translated into executable machine code. *See* PROGRAMMING UTILITIES: DEBUG: A. Within its restrictions, the Assemble command is a handy way of making changes. The Enter Data command, E, could also have been used to change the 15H to a 21H, but the Assemble command is safer, especially for complex instructions. After the new instruction has been entered, press Enter again to stop the assembly process.

There is a danger associated with making changes in memory during debugging: The memory copy of the program is temporary; the changes exist only in memory and when DEBUG exits, they are lost. Changes made to .EXE and .HEX files cannot be written back to disk. To avoid forgetting the changes, write them down. When DEBUG exits, edit the source file *immediately*. Changes made to other files can be written back to disk with DEBUG's Write File or Sectors command, W.

To be sure that the change was made correctly, use the Disassemble (Unassemble) Program command, U, to show the instructions starting at CS:0004H.

```
-U 4  <Enter>
1ACD:0004 B406          MOV   AH,06
1ACD:0006 B2FF          MOV   DL,FF
1ACD:0008 CD21          INT   21
1ACD:000A 740C          JZ    0018
1ACD:000C 3C03          CMP   AL,03
1ACD:000E 7501          JNZ   0011
1ACD:0010 CB            RETF
```

```
1ACD:0011 B401          MOV   AH,01
1ACD:0013 BA0000        MOV   DX,0000
1ACD:0016 CD14          INT   14
1ACD:0018 B403          MOV   AH,03
1ACD:001A BA0000        MOV   DX0000
1ACD:001D CD14          INT   14
1ACD:001F 80E401        AND   AH,01
1ACD:0022 74E0          JZ    0004
```

The change has been correctly made. Now, to test the change, start the program to see if characters make it out the serial port. The problem of data from the serial port not making it to the screen remains, however, so instead of simply starting the program, set a breakpoint at the location in the program that handles incoming serial data (CS:0024H). This technique allows the output section of the code to be tested separately. The breakpoint is set using the Go command, G.

```
-G 24  <Enter>

AX=0130  BX=0000  CX=0131  DX=0000  SP=00FC  BP=0000  SI=0000  DI=0000
DS=1AAD  ES=1AAD  SS=1ABD  CS=1ACD  IP=0024   NV UP EI PL NZ NA PO NC
1ACD:0024 B402          MOV   AH,02
-U  <Enter>
1ACD:0024 B402          MOV   AH,02
1ACD:0026 BA0000        MOV   DX,0000
1ACD:0029 CD14          INT   14
1ACD:002B B406          MOV   AH,06
1ACD:002D CD21          INT   21
1ACD:002F EBD3          JMP   0004
1ACD:0031 0000          ADD   [BX+SI],AL
1ACD:0033 0000          ADD   [BX+SI],AL
1ACD:0035 0000          ADD   [BX+SI],AL
1ACD:0037 0000          ADD   [BX+SI],AL
1ACD:0039 0000          ADD   [BX+SI],AL
1ACD:003B 0000          ADD   [BX+SI],AL
1ACD:003D 0000          ADD   [BX+SI],AL
1ACD:003F 0000          ADD   [BX+SI],AL
1ACD:0041 0000          ADD   [BX+SI],AL
1ACD:0043 0000          ADD   [BX+SI],AL
```

As stated earlier, the serial port is attached to a serial terminal. After execution of the program is started with the Go command, all keys typed on the keyboard are displayed correctly on the terminal, thus confirming the fix made to the INT 21H instruction. To test serial input, a key must be pressed on the terminal, causing the breakpoint at CS:0024H to be executed.

The fact that location CS:0024H was reached indicates that Interrupt 14H is detecting the presence of an input character. To test if the character is now making it to the screen, a breakpoint is needed after the write to the screen. The Disassemble command shows the instructions starting at the current IP value. The program ends at CS:002FH; the instructions shown after that are whatever happened to be in memory when the program was loaded. A good place to set the next breakpoint is CS:002FH, just after the Interrupt 21H call.

```
-G 2f  <Enter>

AX=0600  BX=0000  CX=0131  DX=0000  SP=00FC  BP=0000  SI=0000  DI=0000
DS=1AAD  ES=1AAD  SS=1ABD  CS=1ACD  IP=002F   NV UP EI PL NZ NA PO NC
1ACD:002F EBD3          JMP   0004
```

DEBUG shows that the breakpoint was reached and the character did not print (it should have been on the line after *-G 2f*), so something must be wrong with the Interrupt 21H call. A breakpoint just before the MS-DOS call at CS:002DH should reveal the cause of the problem.

```
-G 2d  <Enter>

AX=0662  BX=0000  CX=0131  DX=0000  SP=00FC  BP=0000  SI=0000  DI=0000
DS=1AAD  ES=1AAD  SS=1ABD  CS=1ACD  IP=002D   NV UP EI PL NZ NA PO NC
1ACD:002D CD21          INT   21
```

The key that was entered on the serial terminal (b) is in AL, where it was returned by Interrupt 14H. Unfortunately, it is not in DL, where it is expected by the Direct Console I/O function (06H) of the MS-DOS command. The MS-DOS function was simply printing a null (00H) and then moving the cursor. An instruction (MOV DL,AL) is missing.

Fixing this problem requires the insertion of a line of code, which is usually difficult to do inside DEBUG. The Move (Copy) Data command, M, can be used to move the code located below the point where the insertion is to be made down 2 bytes, but this will probably throw any subsequent addressing off. It is usually easier to exit DEBUG, edit the source file, and then reassemble. In this case, however, because the instruction to be added is near the last instruction, a patch can easily be made by entering only three instructions: the new one and the two it destroys.

```
-A 2d  <Enter>
1ACD:002D mov dl,al  <Enter>
1ACD:002F int 21  <Enter>
1ACD:0031 jmp 4  <Enter>
1ACD:0033  <Enter>
-U 2b  <Enter>
1ACD:002B B406          MOV   AH,06
1ACD:002D 88C2          MOV   DL,AL
1ACD:002F CD21          INT   21
1ACD:0031 EBD1          JMP   0004
1ACD:0033 0000          ADD   [BX+SI],AL
1ACD:0035 0000          ADD   [BX+SI],AL
1ACD:0037 0000          ADD   [BX+SI],AL
1ACD:0039 0000          ADD   [BX+SI],AL
1ACD:003B 0000          ADD   [BX+SI],AL
1ACD:003D 0000          ADD   [BX+SI],AL
1ACD:003F 0000          ADD   [BX+SI],AL
1ACD:0041 0000          ADD   [BX+SI],AL
1ACD:0043 0000          ADD   [BX+SI],AL
1ACD:0045 0000          ADD   [BX+SI],AL
1ACD:0047 0000          ADD   [BX+SI],AL
1ACD:0049 0000          ADD   [BX+SI],AL
```

The new line of code has been inserted and verified with the Disassemble command. The fix is ready to test. The Trace command could be used to single-step through the program to verify execution. A word of warning is in order, however: The DEBUG Trace command should never be used to trace an Interrupt 21H call. Once the trace enters the MS-DOS call, it will wander around for a while and then lock the machine, requiring a restart. Avoid this problem either by setting a breakpoint just beyond the Interrupt 21H call or by using the Proceed Through Loop or Subroutine command, P. The Proceed command operates in a similar manner to the Trace command but does not trace loops, calls, and interrupts.

Because the fix is fairly certain, use the Go command in its simple form with no breakpoints. The program will execute without further intervention from DEBUG.

```
-G  <Enter>
lasdfgh
Program terminated normally
-Q  <Enter>
```

The *lasdfgh* text entered on the serial terminal is displayed correctly. When a Ctrl-C is entered from the keyboard, the program terminates properly and DEBUG displays the message *Program terminated normally.* Now exit DEBUG with the Quit command, Q.

The source code of TESTCOMM should be edited immediately so that it reflects the two changes made temporarily under DEBUG. Figure 18-10 shows the corrected listing.

```
     TITLE TESTCOMM - TEST COMMSCOP ROUTINE
; ************************************************************************
; *                                                                      *
; *  TESTCOMM                                                            *
; *     THIS ROUTINE PROVIDES DATA FOR THE COMMSCOP ROUTINE.  IT READS   *
; *  CHARACTERS FROM THE KEYBOARD AND WRITES THEM TO COM1 USING          *
; *  INT 14H.  DATA IS ALSO READ FROM INT 14H AND DISPLAYED ON THE       *
; *  SCREEN.  THE ROUTINE RETURNS TO MS-DOS WHEN Ctrl-C IS PRESSED       *
; *  ON THE KEYBOARD.                                                    *
; *                                                                      *
; ************************************************************************

SSEG    SEGMENT PARA STACK 'STACK'
        DW      128 DUP(?)
SSEG    ENDS

CSEG    SEGMENT
        ASSUME  CS:CSEG,SS:SSEG
BEGIN   PROC    FAR
        PUSH    DS                      ;SET UP FOR RET TO MS-DOS
        XOR     AX,AX                   ; .
        PUSH    AX                      ; .
```

Figure 18-10. Correct serial test routine. *(more)*

```
MAINLOOP:
        MOV     AH,6                    ;USE DOS CALL TO CHECK FOR
        MOV     DL,0FFH                 ; KEYBOARD ACTIVITY
        INT     21H                     ; IF NO CHARACTER, JUMP TO
        JZ      TESTCOMM                ; COMM ACTIVITY TEST

        CMP     AL,03                   ;WAS CHARACTER A Ctrl-C?
        JNE     SENDCOMM                ; NO, SEND IT TO SERIAL PORT
        RET                             ; YES, RETURN TO MS-DOS

SENDCOMM:
        MOV     AH,01                   ;USE INT 14H WRITE FUNCTION TO
        MOV     DX,0                    ; SEND DATA TO SERIAL PORT
        INT     14H                     ; .

TESTCOMM:
        MOV     AH,3                    ;GET SERIAL PORT STATUS
        MOV     DX,0                    ; .
        INT     14H                     ; .
        AND     AH,1                    ;ANY DATA WAITING?
        JZ      MAINLOOP                ; NO, GO BACK TO KEYBOARD TEST
        MOV     AH,2                    ;READ SERIAL DATA
        MOV     DX,0                    ; .
        INT     14H                     ; .
        MOV     AH,6                    ;WRITE SERIAL DATA TO SCREEN
        MOV     DL,AL                   ; .
        INT     21H                     ; .
        JMP     MAINLOOP                ;CONTINUE

BEGIN   ENDP
CSEG    ENDS
        END     BEGIN
```

Figure 18-10. Continued.

DEBUG has a rich set of commands and features. The preceding case study shows the more common ones in their most straightforward aspect. Some of the other commands and some useful techniques are described below. *See* PROGRAMMING UTILITIES: DEBUG.

Establishing initial conditions

When a program is loaded for testing, four areas may require initialization:

- Registers
- Data areas
- Default file-control blocks (FCBs)
- Command tail

These areas may also require changes during testing, especially when the programmer is working around bugs or establishing different test conditions.

Registers. Registers are ordinarily set when the program is loaded. The values in them depend on whether a .EXE, .COM, or .HEX file was loaded. Generally, the segment registers, the IP register, and the SP register are set to appropriate values; with the exception of AX, BX, and CX, the rest of the registers are set to zero. BX and CX contain the length of the loaded file. By MS-DOS convention, when a program is loaded, the contents of AL and AH indicate the validity of the drive specifiers in the first and second DEBUG command-line parameters, respectively. Each register contains zero if the corresponding drive was valid, 01H if the drive was valid and wildcards were used, or 0FFH if the drive was invalid.

To change the value of any register, use an alternate form of the Register command. Enter R followed by the two-letter register name. Only 16-bit registers can be changed, so use the X form of the general-purpose registers:

```
-R AX  <Enter>
```

DEBUG will respond with the current contents of the register and prompt for a new value. Either enter a new hexadecimal value or press Enter to keep the current value:

```
AX 0000
:FFFF  <Enter>
```

In this example, the new value of AX is FFFFH.

When changing registers, exercise caution modifying the segment registers. These registers control the execution of the program and should be changed only after careful and thoughtful consideration.

The Register command can also be used to modify the CPU flags.

Data areas. Initializing or changing data areas is easy, and several methods are provided. The Fill Memory command, F, can be used to initialize areas of RAM. For instance,

```
-F 0 L400 0  <Enter>
```

fills DS:0000H through DS:03FFH with zero. (The absence of a segment override causes the Fill command to use its default segment, DS.) Entering

```
-F CS:100 200 1B "[Hello" 0D  <Enter>
```

fills CS:0100H through CS:0200H with many repetitions of the string 1B 5B 48 65 6C 6C 6F 0D. (Note that an address range was specified, not a length.)

When the wholesale changing of memory is not appropriate, the Enter command can be used to edit a small number of locations. The Enter command has two forms: One enters a list of bytes into the specified memory location; the other prompts with the contents of each location and waits for input. Either form can be used as appropriate.

Default file-control blocks and the command tail. The setting of the default FCBs and of the command tail are related functions. When DEBUG is entered, the first parameter following the command DEBUG is the name of the file to be loaded into memory for debugging. If the next two parameters are filenames, FCBs for these files are formatted at

DS:005CH and DS:006CH in the PSP. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: File and Record Management. If either parameter contains a pathname, the corresponding FCB will contain only a valid drive number; the filename field will not be valid. All filenames and switches following the name of the file to be debugged are considered the command tail and are saved in memory starting at DS:0081H. The length of the command tail is in DS:0080H. For example, entering

```
C>DEBUG COMMDUMP.EXE FILE1.DAT FILE2.DAT  <Enter>
```

results in the first FCB (5CH), the second FCB (6CH), and the command tail (81H) being loaded as follows:

```
-D 50  <Enter>
42C9:0050  CD 21 CB 00 00 00 00 00-00 00 00 00 00 46 49 4C   .!...........FIL
42C9:0060  45 31 20 20 20 44 41 54-00 00 00 00 00 46 49 4C   E1   DAT.....FIL
42C9:0070  45 32 20 20 20 44 41 54-00 00 00 00 00 00 00 00   E2   DAT........
42C9:0080  15 20 66 69 6C 65 31 2E-64 61 74 20 66 69 6C 65   . file1.dat file
42C9:0090  32 2E 64 61 74 20 0D 74-20 66 69 6C 65 32 2E 64   2.dat .t file2.d
42C9:00A0  61 74 20 0D 00 00 00 00-00 00 00 00 00 00 00 00   at ............
42C9:00B0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
42C9:00C0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
```

In this example, location DS:005CH contains an unopened FCB for file FILE1.DAT on the current drive. Location DS:006CH contains an unopened FCB for FILE2.DAT on the current drive. (The second FCB cannot be used where it is and must be moved to another location before the first FCB is opened.) Location DS:0080H contains the length of the command tail, 15H (21) bytes. The next 21 bytes are the command tail prepared by DEBUG; they correspond exactly to what the command tail would be if the program had been loaded by COMMAND.COM instead of by DEBUG.

The default FCBs and the command tail can also be set after the program has been loaded, by using the Name File or Command-Tail Parameters command, N. DEBUG treats the string of characters that follow the Name command as the command tail: If the first two parameters are filenames, they become the first and second FCBs, respectively. The Name command also places the string at DS:0081H, with the length of the string at DS:0080H. Entering the DEBUG command

```
-N FILE1.DAT FILE2.DAT  <Enter>
```

produces the same results as specifying the filenames in the command line. When employed in this manner, the Name command is useful for initializing command-tail data that was not in the command line or for changing the command-tail data to test different aspects of a program. (If files are named in this manner, they are not validated until the Load File or Sectors command, L, is used.) Note that the data following the Name command need not be filenames; it can be any parameters, data, or switches that the application program expects to see.

More on breakpoints

The case study at the beginning of this section used breakpoints in their simplest form: Only a single breakpoint was specified at a time and the execution address was considered to be the current IP. The Go command is also capable of setting multiple breakpoints and of beginning execution at any address in memory. The more general form of the Go command is

G[=*address*] [*address* [*address*...]]

If Go is used with no operands, execution begins at the current value of CS:IP and no breakpoints are set. If the =*address* operand is used, DEBUG sets IP to the address specified and execution then begins at the new CS:IP. The other optional addresses are breakpoints. When execution reaches one of these breakpoints, DEBUG stops and displays the system's registers. As many as 10 breakpoints can be set on one Go command, and they can be in any order.

The breakpoint addresses must be on instruction boundaries because DEBUG replaces the instruction at each breakpoint address with an INT 03H instruction (0CCH). DEBUG saves the replaced instructions internally. When any breakpoint is reached, DEBUG stops execution and restores the instructions at *all* the breakpoints; if no breakpoint is reached, the instructions are not restored and the Load command must be used to reload the original program.

The multiple-breakpoint feature of the Go command allows the tracing of program execution when branches exist in the code. When a program contains, for instance, a conditional jump on the zero flag, a breakpoint can be placed in each of the two possible branches. When the branch is reached, one of the two breakpoints will be encountered shortly thereafter. When DEBUG displays the breakpoint, the programmer knows which branch was taken. Moving through a program with breakpoints at key locations is faster than using the Trace command to execute each and every instruction.

Multiple breakpoints can also be used to home in on a bad piece of code. This technique is particularly useful in those nasty situations when there are no symptoms except that the system locks up and must be restarted. When debugging a problem such as this, set breakpoints at each of the major sections of the program and then note those breakpoints that are executed successfully, continuing until the system locks up. The problem lies somewhere between the last successful breakpoint and the next breakpoint set. Now repeat the processes, setting breakpoints between the last breakpoint and the one that was never reached. By progressively narrowing the gap between breakpoints, the exact offending instruction can be isolated.

Some general comments about the Go command and breakpoints:

- After a program has reached completion and returned to MS-DOS, it must be reloaded with the Load command before it can be executed again. (DEBUG intercepts this return and displays *Program terminated normally.*)
- Because DEBUG replaces program instructions with an INT 03H instruction to form breakpoints, the break address must be on an instruction boundary. If it is not, the INT 03H will be stuck in the middle of an instruction, causing strange and sometimes entertaining results.

- Breakpoints cannot be set in data, because data is not executed.
- The target program's SS:SP registers must point to a valid stack that has at least 6 bytes of stack space available. When the Go command is executed, it pushes the target program's flags and CS and IP registers onto the stack and then transfers control to the program with an IRET instruction. Thus, if the target program's stack is not valid or is too small, the system may crash.
- Finally, and obviously, breakpoints cannot be set in read-only memory (the ROM BIOS, for instance).

Using the Write commands

After a program has been debugged, fixed, and tested with DEBUG, the temptation exists to write the patched program directly back to the disk as a .COM file. This action is sometimes legitimate, but only rarely. The technique will be explained in a moment, but first a sermon:

DON'T DO IT.

One of the greatest sadnesses in a programmer's life comes when, after a program has been running wonderfully, enhancements are made to the source code and the recompiled program suddenly has bugs in it that haven't been seen for months. Always make any debugging patches permanent in the source file immediately.

Unless, of course, the source code is not available. This is the only time saving a patched program is permissible. For example, sometimes commercial programs require patching because the program does not quite fit the hardware it must run on or because bugs have been found in the program. The source of these patches is sometimes word-of-mouth, sometimes a bulletin-board service, and sometimes the program's manufacturer.

Even when legitimate reasons exist to save patched code, precautions should be taken. Be very careful, meticulous, and alert as the patches are applied. Understand each step before undertaking it. Most important of all, always have a backup of the original unpatched program safely on a floppy disk.

Use the Write command to write the program image to disk. A starting address can optionally be specified; otherwise the write starts at CS:0100H. The name of the file will be either the name specified in the last Name command or the name of the program from the DEBUG command line if the Name command has not been used. The number of bytes to be written is in BX and CX, with the most significant half in BX. These registers will have been loaded correctly when the program was loaded, but they should be checked if the program has executed since it was loaded.

The .EXE and .HEX file types cannot be written to disk with the Write command. The command performs no formatting and only writes the binary image of memory to the disk file. Thus, all programs written with Write must be .COM files. The image of a .EXE or .HEX file can still be written as a .COM file provided no segment fixups are required and provided the other rules for a .COM file are followed. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: Structure of an Application Program. (A segment fixup is a segment address that must be provided by the loader when the

program is originally loaded. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING TOOLS: Object Modules.) If a .EXE file containing a segment fixup is written as a .COM file, the new file will execute correctly only when loaded at exactly the same address as the original file, and this is difficult to ensure for programs running under MS-DOS.

If it is necessary to patch a .EXE or .HEX file and the exact addresses relative to the start of the file are known, use the following procedure:

1. Rename (or better yet, copy) the file to an extension other than .EXE or .HEX.
2. Load the program image into memory by placing the new name on DEBUG's command line. Note that the loaded file is an image of the disk file and is not executable.
3. Modify the program image in memory, but *never* try to execute the program. Results would be unpredictable and the program image could be damaged.
4. Write the modified image back to disk using a simple *w*. No other action is needed, because the original load will have set the filename and the correct length in BX and CX.
5. Rename the file to a name with the correct .EXE or .HEX extension. The new name need not be the same as the original, but it should have the same extension.

The same technique can be used to load, modify, and save data files. Simply make sure that the file does not have an extension of .COM, .EXE, or .HEX. The data file will be loaded at address CS:0100H. (DEBUG treats the file much the same as a .COM file.) After patching the data (the Enter command works best), use the Write command to write it back to the disk.

SYMDEB

SYMDEB is an extension of DEBUG; virtually all the DEBUG commands and techniques still work as expected. The major new feature, and the source of the name SYMDEB, is symbolic debugging: SYMDEB can use all public labels in a program for reference, instead of using hexadecimal offset addresses. In addition, SYMDEB allows the use of line numbers for reference in compatible high-order languages; source-line display within SYMDEB is also possible for these languages. Currently, the languages supporting these options are Microsoft FORTRAN versions 3.0 and later, Microsoft Pascal versions 3.0 and later, and Microsoft C versions 2.0 and later. Versions 4.0 and earlier of the Microsoft Macro Assembler (MASM) do not generate the data needed for line-number display and source-line debugging.

In addition to symbolic debugging, SYMDEB has added several other new features and has expanded existing DEBUG features:

- Breakpoints have been made more sophisticated with the addition of "sticky" breakpoints. Unlike the breakpoints set with the Go command, sticky breakpoints remain attached to the program throughout a SYMDEB session until they are explicitly removed. Specific commands are supplied for listing, removing, enabling, and disabling sticky breakpoints.
- DEBUG's Display Memory command, D, has been extended so that data can be displayed in different formats.

- Full redirection is supported.
- A stack trace feature has been added.
- Terminate-and-stay-resident programs are supported.
- A shell escape command has been added to allow the execution of MS-DOS commands and programs without leaving SYMDEB and the debugging session.

These additions allow more sophisticated debugging techniques to be used and, in some cases, also simplify locating problems. To see the advantages of using symbols and sticky breakpoints in debugging, consider a type of program that is one of the most difficult to debug—the TSR.

Debugging TSRs with SYMDEB

Terminate-and-stay-resident routines can be difficult to debug. They exist in two worlds and can have bugs associated with each. At the outset, they are usually simple programs that perform some initialization task and then exit. At this point, they are transformed into another type of beast entirely—resident routines that are more a part of the operating system than of any application program. Each form of the program must be debugged separately, using different techniques.

The TSR routine used for this case study is the same one created previously to serve as external instrumentation to trace serial communications. The program was called COMMSCOP, but to avoid confusion of that working program with the broken one presented here, the name has been changed to BADSCOP. BADSCOP was assembled and linked in the usual manner and then converted to a .COM file using EXE2BIN. When it was installed, it returned normally, but at the first attempt to issue an Interrupt 14H, the system locked up completely. Warm booting was not sufficient to restore it, and a power-on cold boot was required to get the system working again.

Figure 18-11 is a listing of BADSCOP. The only difference from COMMSCOP, aside from the errors, is the addition of two PUBLIC statements to make all the procedure names and the important data names available to SYMDEB.

```
        TITLE   BADSCOP - BAD VERSION OF COMMUNICATIONS TRACE UTILITY
; ********************************************************************
; *                                                                  *
; *  BADSCOP -                                                       *
; *     THIS PROGRAM MONITORS THE ACTIVITY ON A SPECIFIED COMM PORT  *
; *     AND PLACES A COPY OF ALL COMM ACTIVITY IN A RAM BUFFER.  EACH*
; *     ENTRY IN THE BUFFER IS TAGGED TO INDICATE WHETHER THE BYTE   *
; *     WAS SENT BY OR RECEIVED BY THE SYSTEM.                       *
; *                                                                  *
; *     BADSCOP IS INSTALLED BY ENTERING                             *
; *                                                                  *
; *                     BADSCOP                                      *
; *                                                                  *
```

Figure 18-11. An incorrect version of the serial trace utility. *(more)*

```
; *  THIS WILL INSTALL BADSCOP AND SET UP A 64K BUFFER TO BE USED       *
; *  FOR DATA LOGGING.  REMEMBER THAT 2 BYTES ARE REQUIRED FOR         *
; *  EACH COMM BYTE, SO THE BUFFER IS ONLY 32K EVENTS LONG, OR ABOUT   *
; *  30 SECONDS OF CONTINUOUS 9600 BAUD DATA.  IN THE REAL WORLD,      *
; *  ASYNC DATA IS RARELY CONTINUOUS, SO THE BUFFER WILL PROBABLY      *
; *  HOLD MORE THAN 30 SECONDS WORTH OF DATA.                          *
; *                                                                    *
; *  WHEN INSTALLED, BADSCOP INTERCEPTS ALL INT 14H CALLS.  IF THE     *
; *  PROGRAM HAS BEEN ACTIVATED AND THE INT IS EITHER SEND OR RE-      *
; *  CEIVE DATA, A COPY OF THE DATA BYTE, PROPERLY TAGGED, IS PLACED   *
; *  IN THE BUFFER.  IN ANY CASE, DATA IS PASSED ON TO THE REAL        *
; *  INT 14H HANDLER.                                                  *
; *                                                                    *
; *  BADSCOP IS INVOKED BY ISSUING AN INT 60H CALL.  THE INT HAS       *
; *  THE FOLLOWING CALLING SEQUENCE:                                   *
; *                                                                    *
; *       AH - COMMAND                                                 *
; *            0 - STOP TRACING, PLACE STOP MARK IN BUFFER             *
; *            1 - FLUSH BUFFER AND START TRACE                        *
; *            2 - RESUME TRACE                                        *
; *            3 - RETURN COMM BUFFER ADDRESSES                        *
; *       DX - COMM PORT (ONLY USED WITH AH = 1 or 2)                  *
; *            0 - COM1                                                *
; *            1 - COM2                                                *
; *                                                                    *
; *  THE FOLLOWING DATA IS RETURNED IN RESPONSE TO AH = 3:             *
; *                                                                    *
; *       CX - BUFFER COUNT IN BYTES                                   *
; *       DX - SEGMENT ADDRESS OF THE START OF THE BUFFER              *
; *       BX - OFFSET ADDRESS OF THE START OF THE BUFFER               *
; *                                                                    *
; *  THE COMM BUFFER IS FILLED WITH 2-BYTE DATA ENTRIES OF THE         *
; *  FOLLOWING FORM:                                                   *
; *                                                                    *
; *       BYTE 0 - CONTROL                                             *
; *            BIT 0 - ON FOR RECEIVED DATA, OFF FOR TRANS.            *
; *            BIT 7 - STOP MARK - INDICATES COLLECTION WAS            *
; *                  INTERRUPTED AND RESUMED.                          *
; *       BYTE 1 - 8-BIT DATA                                          *
; *                                                                    *
; ********************************************************************

        PUBLIC  INITIALIZE,CONTROL,VECTOR_INIT,COMMSCOPE
        PUBLIC  OLD_COMM_INT,COUNT,STATUS,PORT,BUFPNTR

CSEG    SEGMENT
        ASSUME  CS:CSEG,DS:CSEG
        ORG     100H                    ;TO MAKE A COM FILE
```

Figure 18-11. Continued. *(more)*

```
INITIALIZE:
        JMP     VECTOR_INIT             ;JUMP TO THE INITIALIZATION
                                        ; ROUTINE WHICH, TO SAVE SPACE,
                                        ; IS IN THE COMM BUFFER

;
;  SYSTEM VARIABLES
;
OLD_COMM_INT    DD      ?               ;ADDRESS OF REAL COMM INT
COUNT           DW      0               ;BUFFER COUNT
COMMSCOPE_INT   EQU     60H             ;COMMSCOPE CONTROL INT
STATUS          DB      0               ;PROCESSING STATUS
                                        ; 0 - OFF
                                        ; 1 - ON
PORT            DB      0               ;COMM PORT BEING TRACED
BUFPNTR         DW      VECTOR_INIT     ;NEXT BUFFER LOCATION

        SUBTTL  DATA INTERRUPT HANDLER
PAGE
; ************************************************************************
; *                                                                      *
; *  COMMSCOPE                                                           *
; *  THIS PROCEDURE INTERCEPTS ALL INT 14H CALLS AND LOGS THE DATA       *
; *  IF APPROPRIATE.                                                     *
; *                                                                      *
; ************************************************************************
COMMSCOPE       PROC    NEAR

        TEST    CS:STATUS,1             ;ARE WE ON?
        JZ      OLD_JUMP                ; NO, SIMPLY JUMP TO OLD HANDLER

        CMP     AH,00H                  ;SKIP SETUP CALLS
        JE      OLD_JUMP                ; .

        CMP     AH,03H                  ;SKIP STATUS REQUESTS
        JAE     OLD_JUMP                ; .

        CMP     AH,02H                  ;IS THIS A READ REQUEST?
        JE      GET_READ                ; YES, GO PROCESS

;
;  DATA WRITE REQUEST - SAVE IF APPROPRIATE
;
        CMP     DL,CS:PORT              ;IS WRITE FOR PORT BEING TRACED?
        JNE     OLD_JUMP                ; NO, JUST PASS IT THROUGH

        PUSH    DS                      ;SAVE CALLER'S REGISTERS
        PUSH    BX                      ; .
        PUSH    CS                      ;SET UP DS FOR OUR PROGRAM
        POP     DS                      ; .
        MOV     BX,BUFPNTR              ;GET ADDRESS OF NEXT BUFFER LOCATION
```

Figure 18-11. Continued. *(more)*

```
        MOV     [BX],BYTE PTR 0         ;MARK AS TRANSMITTED BYTE
        MOV     [BX+1],AL               ;SAVE DATA IN BUFFER
        INC     COUNT                   ;INCREMENT BUFFER BYTE COUNT
        INC     COUNT                   ; .
        INC     BX                      ;POINT TO NEXT LOCATION
        INC     BX                      ; .
        MOV     BUFPNTR,BX              ;SAVE NEW POINTER
        JNZ     WRITE_DONE              ;ZERO INDICATES BUFFER HAS WRAPPED

        MOV     STATUS,0                ;TURN COLLECTION OFF - BUFFER FULL
WRITE_DONE:
        POP     BX                      ;RESTORE CALLER'S REGISTERS
        POP     DS                      ; .
        JMP     OLD_JUMP                ;PASS REQUEST ON TO BIOS ROUTINE
;
;  PROCESS A READ DATA REQUEST AND WRITE TO BUFFER IF APPROPRIATE
;
GET_READ:
        CMP     DL,CS:PORT              ;IS READ FOR PORT BEING TRACED?
        JNE     OLD_JUMP                ; NO, JUST PASS IT THROUGH

        PUSH    DS                      ;SAVE CALLER'S REGISTERS
        PUSH    BX                      ; .
        PUSH    CS                      ;SET UP DS FOR OUR PROGRAM
        POP     DS                      ; .

        PUSHF                           ;FAKE INT 14H CALL
        CLI                             ; .
        CALL    OLD_COMM_INT            ;PASS REQUEST ON TO BIOS
        TEST    AH,80H                  ;VALID READ?
        JNZ     READ_DONE               ; NO, SKIP BUFFER UPDATE

        MOV     BX,BUFPNTR              ;GET ADDRESS OF NEXT BUFFER LOCATION
        MOV     [BX],BYTE PTR 1         ;MARK AS RECEIVED BYTE
        MOV     [BX+1],AL               ;SAVE DATA IN BUFFER
        INC     COUNT                   ;INCREMENT BUFFER BYTE COUNT
        INC     COUNT                   ; .
        INC     BX                      ;POINT TO NEXT LOCATION
        INC     BX                      ; .
        MOV     BUFPNTR,BX              ;SAVE NEW POINTER
        JNZ     READ_DONE               ;ZERO INDICATES BUFFER HAS WRAPPED

        MOV     STATUS,0                ;TURN COLLECTION OFF - BUFFER FULL
READ_DONE:
        POP     BX                      ;RESTORE CALLER'S REGISTERS
        POP     DS                      ; .
        IRET

;
;  JUMP TO COMM BIOS ROUTINE
;
OLD_JUMP:
        JMP     OLD_COMM_INT

COMMSCOPE ENDP
```

Figure 18-11. Continued. *(more)*

```
        SUBTTL  CONTROL INTERRUPT HANDLER
PAGE
; ************************************************************************
; *                                                                      *
; *  CONTROL                                                             *
; *     THIS ROUTINE PROCESSES CONTROL REQUESTS.                         *
; *                                                                      *
; ************************************************************************

CONTROL PROC    NEAR
        CMP     AH,00H                  ;STOP REQUEST?
        JNE     CNTL_START              ; NO, CHECK START
        PUSH    DS                      ;SAVE REGISTERS
        PUSH    BX                      ; .
        PUSH    CS                      ;SET DS FOR OUR ROUTINE
        POP     DS
        MOV     STATUS,0                ;TURN PROCESSING OFF
        MOV     BX,BUFPNTR              ;PLACE STOP MARK IN BUFFER
        MOV     [BX],BYTE PTR 80H       ; .
        MOV     [BX+1],BYTE PTR 0FFH    ; .
        INC     COUNT                   ;INCREMENT COUNT
        INC     COUNT                   ; .
        POP     BX                      ;RESTORE REGISTERS
        POP     DS                      ; .
        JMP     CONTROL_DONE

CNTL_START:
        CMP     AH,01H                  ;START REQUEST?
        JNE     CNTL_RESUME             ; NO, CHECK RESUME
        MOV     CS:PORT,DL              ;SAVE PORT TO TRACE
        MOV     CS:BUFPNTR,OFFSET VECTOR_INIT   ;RESET BUFFER TO START
        MOV     CS:COUNT,0              ;ZERO COUNT
        MOV     CS:STATUS,1             ;START LOGGING
        JMP     CONTROL_DONE

CNTL_RESUME:
        CMP     AH,02H                  ;RESUME REQUEST?
        JNE     CNTL_STATUS             ; NO, CHECK STATUS
        CMP     CS:BUFPNTR,0            ;END OF BUFFER CONDITION?
        JE      CONTROL_DONE            ; YES, DO NOTHING
        MOV     CS:PORT,DL              ;SAVE PORT TO TRACE
        MOV     CS:STATUS,1             ;START LOGGING
        JMP     CONTROL_DONE

CNTL_STATUS:
        CMP     AH,03H                  ;RETURN STATUS REQUEST?
        JNE     CONTROL_DONE            ; NO, ERROR - DO NOTHING
        MOV     CX,CS:COUNT             ;RETURN COUNT
        PUSH    CS                      ;RETURN SEGMENT ADDR OF BUFFER
        POP     DX                      ; .
        MOV     BX,OFFSET VECTOR_INIT   ;RETURN OFFSET ADDR OF BUFFER
```

Figure 18-11. Continued. *(more)*

```
CONTROL_DONE:
        IRET

CONTROL ENDP


        SUBTTL  INITIALIZE INTERRUPT VECTORS
PAGE
; ************************************************************************
; *                                                                      *
; *  VECTOR_INIT                                                         *
; *  THIS PROCEDURE INITIALIZES THE INTERRUPT VECTORS AND THEN           *
; *  EXITS VIA THE MS-DOS TERMINATE-AND-STAY-RESIDENT FUNCTION.          *
; *  A BUFFER OF 64K IS RETAINED.  THE FIRST AVAILABLE BYTE              *
; *  IN THE BUFFER IS THE OFFSET OF VECTOR_INIT.                         *
; *                                                                      *
; ************************************************************************

        EVEN                            ;ASSURE BUFFER ON EVEN BOUNDARY
VECTOR_INIT     PROC    NEAR
;
;  GET ADDRESS OF COMM VECTOR (INT 14H)
;
        MOV     AH,35H
        MOV     AL,14H
        INT     21H
;
;  SAVE OLD COMM INT ADDRESS
;
        MOV     WORD PTR OLD_COMM_INT,BX
        MOV     AX,ES
        MOV     WORD PTR OLD_COMM_INT[2],AX
;
;  SET UP COMM INT TO POINT TO OUR ROUTINE
;
        MOV     DX,OFFSET COMMSCOPE
        MOV     AH,25H
        MOV     AL,14H
        INT     21H
;
;  INSTALL CONTROL ROUTINE INT
;
        MOV     DX,OFFSET CONTROL
        MOV     AH,25H
        MOV     AL,COMMSCOPE_INT
        INT     21H
;
;  SET LENGTH TO 64K, EXIT AND STAY RESIDENT
;
        MOV     AX,3100H                ;TERM AND STAY RES COMMAND
        MOV     DX,1000H                ;64K RESERVED
        INT     21H                     ;DONE
```

Figure 18-11. Continued. *(more)*

```
VECTOR_INIT ENDP

CSEG    ENDS
        END     INITIALIZE
```

Figure 18-11. Continued.

In order to use the symbolic debugging features of SYMDEB, a symbol file must be built in a specific format. The SYMDEB utility MAPSYM performs this function, using the contents of the .MAP file built by LINK. MAPSYM is easy to use because it has only two parameters: the .MAP file and the /L switch (which triggers verbose mode). The symbol table for BADSCOP is built as follows:

```
C>MAPSYM BADSCOP  <Enter>
```

This operation produces a symbol file called BADSCOP.SYM.

Armed with the .SYM file and the usual collection of listing and design notes, the programmer can begin the debugging process using SYMDEB.

The first task is to discover if the BADSCOP TSR is installing correctly. To test this, run the .COM file under SYMDEB by typing

```
C>SYMDEB BADSCOP.SYM BADSCOP.COM  <Enter>
```

Note the order in which operands are passed to SYMDEB — it is not the order that would be expected. All switches (none were used here) must immediately follow the word *SYMDEB*. These switches must be followed in turn by the fully qualified names of any symbol files (in this case, BADSCOP.SYM). Only then is the name of the file to be debugged given. If BADSCOP expected any parameters in the command tail, they would be last. This potential need for command-tail data is the reason the name of the file to be debugged follows the name of the symbol file. SYMDEB knows that the first non-.SYM file it encounters is the file to be loaded; the parameters that follow the filename may be of any form and number.

When SYMDEB begins, it displays

```
Microsoft (R) Symbolic Debug Utility  Version 4.00
Copyright (C) Microsoft Corp 1984, 1985. All rights reserved.

Processor is [80286]
```

The debugger identifies itself and then notes the type of CPU it is running on — in this case, an Intel 80286. The Display or Modify Registers command, R, gives the same display that DEBUG gives, with one exception.

```
-R  <Enter>
AX=0000  BX=0000  CX=0133  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=1FD0  ES=1FD0  SS=1FD0  CS=1FD0  IP=0100   NV UP EI PL NZ NA PO NC
CSEG:INITIALIZE:
1FD0:0100 E90701         JMP  VECTOR_INIT
```

The instruction at CS:IP, JMP, is now preceded by the information that the instruction is at label *INITIALIZE* within segment *CSEG*. An examination of Figure 18-11 shows that this is indeed the case.

To check that all the symbols requested with the PUBLIC statement are present, use the X?* form of the Examine Symbol Map command.

```
-X?*  <Enter>

CSEG: (1FD0)
0100 INITIALIZE   0103 OLD_COMM_INT 0107 COUNT         0109 STATUS
010A PORT         010B BUFPNTR      010D COMMSCOPE     018F CONTROL
020A VECTOR_INIT
```

The display shows that the value of *CSEG* (1FD0H) matches the current value of CS. The offset values shown for the procedure names and data names match the numbers from an assembled listing. Because this is a .COM file, there is only one segment. If there had been other segments — a data segment, for instance — they would have been shown with their values and associated labels and offsets.

The purpose of this test is to determine whether the problems this program is having are caused by an incorrect installation. First, use the Trace Program Execution command, T, to trace through the first few steps.

```
-T7  <Enter>
AX=0000  BX=0000  CX=0133  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=1FD0  ES=1FD0  SS=1FD0  CS=1FD0  IP=020A   NV UP EI PL NZ NA PO NC
CSEG:VECTOR_INIT:
1FD0:020A B435           MOV  AH,35                          ;'5'
AX=3500  BX=0000  CX=0133  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=1FD0  ES=1FD0  SS=1FD0  CS=1FD0  IP=020C   NV UP EI PL NZ NA PO NC
1FD0:020C B014           MOV  AL,14
AX=3514  BX=0000  CX=0133  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=1FD0  ES=1FD0  SS=1FD0  CS=1FD0  IP=020E   NV UP EI PL NZ NA PO NC
1FD0:020E CD21           INT  21  ;Get Interrupt Vector
AX=3514  BX=1375  CX=0133  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=1FD0  ES=1567  SS=1FD0  CS=1FD0  IP=0210   NV UP EI PL NZ NA PO NC
1FD0:0210 891E0301       MOV  [OLD_COMM_INT],BX                DS:0103=0000
AX=3514  BX=1375  CX=0133  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=1FD0  ES=1567  SS=1FD0  CS=1FD0  IP=0214   NV UP EI PL NZ NA PO NC
1FD0:0214 8CC0           MOV  AX,ES
AX=1567  BX=1375  CX=0133  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=1FD0  ES=1567  SS=1FD0  CS=1FD0  IP=0216   NV UP EI PL NZ NA PO NC
1FD0:0216 A30501         MOV  [OLD_COMM_INT+02 (0105)],AX        DS:0105=0000
AX=1567  BX=1375  CX=0133  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=1FD0  ES=1567  SS=1FD0  CS=1FD0  IP=0219   NV UP EI PL NZ NA PO NC
1FD0:0219 BA0D01         MOV  DX,010D
```

This part of the program uses Interrupt 21H Function 35H to obtain the current vector for Interrupt 14H. Note that, unlike DEBUG, SYMDEB coasts right through an Interrupt 21H call with no problems. It not only knows enough not to make the call but also displays the type of function call being made, based on the value in AH.

To make sure that the correct vector for the old Interrupt 14H handler has been stored, use the Display Doublewords command, DD, in conjunction with a symbol name.

```
-DD OLD_COMM_INT L1  <Enter>
1FD0:01030  1567:1375
```

This is the correct vector address (1567:1375H). Now trace through the next part of the program, which establishes the new vectors for interrupts.

```
-T8  <Enter>
AX=1567  BX=1375  CX=0133  DX=010D  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=1FD0  ES=1567  SS=1FD0  CS=1FD0  IP=021C   NV UP EI PL NZ NA PO NC
1FD0:021C B425          MOV  AH,25                          ;'%'
AX=2567  BX=1375  CX=0133  DX=010D  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=1FD0  ES=1567  SS=1FD0  CS=1FD0  IP=021E   NV UP EI PL NZ NA PO NC
1FD0:021E B014          MOV  AL,14
AX=2514  BX=1375  CX=0133  DX=010D  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=1FD0  ES=1567  SS=1FD0  CS=1FD0  IP=0220   NV UP EI PL NZ NA PO NC
1FD0:0220 CD21          INT  21  ;Set Vector
AX=2514  BX=1375  CX=0133  DX=010D  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=1FD0  ES=1567  SS=1FD0  CS=1FD0  IP=0222   NV UP EI PL NZ NA PO NC
1FD0:0222 BA8F01        MOV  DX,018F
AX=2514  BX=1375  CX=0133  DX=018F  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=1FD0  ES=1567  SS=1FD0  CS=1FD0  IP=0225   NV UP EI PL NZ NA PO NC
1FD0:0225 B425          MOV  AH,25                          ;'%'
AX=2514  BX=1375  CX=0133  DX=018F  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=1FD0  ES=1567  SS=1FD0  CS=1FD0  IP=0227   NV UP EI PL NZ NA PO NC
1FD0:0227 B060          MOV  AL,60                          ;'`'
AX=2560  BX=1375  CX=0133  DX=018F  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=1FD0  ES=1567  SS=1FD0  CS=1FD0  IP=0229   NV UP EI PL NZ NA PO NC
1FD0:0229 CD21          INT  21  ;Set Vector
AX=2560  BX=1375  CX=0133  DX=018F  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=1FD0  ES=1567  SS=1FD0  CS=1FD0  IP=022B   NV UP EI PL NZ NA PO NC
1FD0:022B B80031        MOV  AX,3100
```

Examination of these trace steps shows that all went normally. The new Interrupt 14H vector has been established at *COMMSCOPE*; the vector for the new Interrupt 60H has also been correctly installed. Use the Go command, G, to allow the program to continue to termination and then use the Quit command, Q, to exit SYMDEB.

```
-G  <Enter>

Program terminated and stayed resident (0)
-Q  <Enter>
```

SYMDEB displays the information that the program terminated with a completion code of zero and stayed resident. This is as it should be, and the conclusion is that the installation portion of this TSR is running properly. The problem must be in the real-time execution of the program.

Debugging the resident portion of a TSR is complicated but not especially difficult. A simple program is used to exercise the TSR, and it is this program that is debugged. As this driver program exercises the TSR, the tracing process continues into the resident routine.

Because symbol tables exist for the TSR, symbolic debugging can be used to follow its execution.

The driver program will be TESTCOMM, shown in Figure 18-10. To make the program more easily usable by SYMDEB, one line has been added before the first SEGMENT statement:

```
PUBLIC      BEGIN,MAINLOOP,SENDCOMM,TESTCOMM
```

Using the .MAP file produced by LINK, the MAPSYM routine creates TESTCOMM.SYM. TESTCOMM can now be invoked with two symbol files:

```
C>SYMDEB TESTCOMM.SYM BADSCOP.SYM TESTCOMM.EXE  <Enter>
```

SYMDEB will load both symbol files and then load TESTCOMM.EXE. Because the name of the TESTCOMM.SYM file matches the name of the program being loaded, SYMDEB makes TESTCOMM.SYM the active symbol file.

Use the Register command to show that the test program was properly loaded.

```
-R  <Enter>
AX=0000  BX=0000  CX=0133  DX=0000  SP=0100  BP=0000  SI=0000  DI=0000
DS=38EE  ES=38EE  SS=38FE  CS=390E  IP=0000   NV UP EI PL NZ NA PO NC
CSEG:BEGIN:
390E:0000 1E             PUSH     DS
```

Then use the Examine Symbol Map command to determine whether the symbol files were loaded correctly. The form X* lists all the symbol maps and their segments; the form X?* lists all the symbols for the current symbol map and segment.

```
-X*  <Enter>
[38FE TESTCOMM]
      [390E CSEG]
 0000 BADSCOP
       0000 CSEG
-X?*  <Enter>

CSEG: (390E)
0000 BEGIN    0004 MAINLOOP 0011 SENDCOMM 0018 TESTCOMM
```

The current symbol map and segment are shown in square brackets. The symbol map for BADSCOP is also present but not selected. Note that there are no values associated with BADSCOP in the listing produced by the X?* command, because all the symbols currently available to SYMDEB are shown and only the symbols in TESTCOMM's *CSEG* are available (that is, TESTCOMM.SYM is the only active symbol file).

Recall that the BADSCOP TSR loaded normally but locked the system up at the first attempt to issue an Interrupt 14H. This behavior indicates that the problem is associated with an Interrupt 14H call. TESTCOMM repeatedly makes the system fail, but which of the Interrupt 14H calls within TESTCOMM is causing the trouble is not known. The most straightforward approach would be to put a breakpoint just before each Interrupt 14H instruction. Use the Disassemble (Unassemble) command, U, to find the location of all Interrupt 14H calls.

```
-U MAINLOOP L19  <Enter>
CSEG:MAINLOOP:
390E:0004 B406          MOV  AH,06
390E:0006 B2FF          MOV  DL,FF
390E:0008 CD21          INT  21
390E:000A 740C          JZ   TESTCOMM
390E:000C 3C03          CMP  AL,03
390E:000E 7501          JNZ  SENDCOMM
390E:0010 CB            RETF
CSEG:SENDCOMM:
390E:0011 B401          MOV  AH,01
390E:0013 BA0000        MOV  DX,BADSCOP!CSEG
390E:0016 CD14          INT  14
CSEG:TESTCOMM:
390E:0018 B403          MOV  AH,03
390E:001A BA0000        MOV  DX,BADSCOP!CSEG
390E:001D CD14          INT  14
390E:001F 80E401        AND  AH,01
390E:0022 74E0          JZ   MAINLOOP
390E:0024 B402          MOV  AH,02
390E:0026 BA0000        MOV  DX,BADSCOP!CSEG
390E:0029 CD14          INT  14
390E:002B B406          MOV  AH,06
390E:002D 8AD0          MOV  DL,AL
390E:002F CD21          INT  21
390E:0031 EBD1          JMP  MAINLOOP
```

The Disassemble request starts at *MAINLOOP* and acts on the next 25 (19H) instructions. SYMDEB displays symbol names instead of numbers whenever it can. However, it does get confused from time to time, so a grain of salt might be needed when reading the disassembly. Notice, for instance, the MOV DX,0 instructions at offsets 13H, 1AH, and 26H. SYMDEB has decided that what is being moved is not zero, but BADSCOP!CSEG. (The ! identifies a mapname in the same way a : defines a segment.) In this case, SYMDEB searched its map tables for an address of zero and found one at *CSEG* in BADSCOP. This segment has the address of zero because it has not been initialized.

Ignoring the name confusions, the disassembly clearly shows the three INT 14H instructions at offsets 16H, 1DH, and 29H. Use the Set Breakpoints command, BP, to set a sticky, or permanent, breakpoint at each of these locations. In this way, any Interrupt 14H call issued by TESTCOMM will be intercepted before it executes. Use the List Breakpoints command, BL, to verify the breakpoints.

```
-BP 16  <Enter>
-BP 1D  <Enter>
-BP 29  <Enter>
-BL  <Enter>
0 e 390E:0016 [CSEG:SENDCOMM+05 (0016)]
1 e 390E:001D [CSEG:TESTCOMM+05 (001D)]
2 e 390E:0029 [CSEG:TESTCOMM+11 (0029)]
```

The List Breakpoints command shows that breakpoint 0 is enabled and set to *SENDCOMM+05*, or CS:0016H. Likewise, breakpoint 1 is at CS:001DH and breakpoint 2 is at CS:0029H. It is important to trap on an Interrupt 14H so that the subsequent actions of the Interrupt 14H service routine can be traced. Now allow the program to execute until it encounters a breakpoint.

```
-G  <Enter>
AX=0300  BX=0000  CX=0133  DX=0000  SP=00FC  BP=0000  SI=0000  DI=0000
DS=38EE  ES=38EE  SS=38FE  CS=390E  IP=001D   NV UP EI PL ZR NA PE NC
390E:001D CD14           INT     14                             ;BR1
```

The first Interrupt 14H encountered is the one at the second breakpoint, breakpoint 1, as can be seen from the address at which execution broke. Also, SYMDEB was kind enough to include the comment *;BR1* on the disassembled line, indicating that this is Break Request 1. The instruction at this location is a request for serial port status (AH = 3) and the registers are loaded correctly. Execution can now be passed to the TSR by simply executing the current instruction. (Remember that the instruction displayed at a breakpoint has not yet been executed.)

```
-T  <Enter>
AX=0300  BX=0000  CX=0133  DX=0000  SP=00F6  BP=0000  SI=0000  DI=00Q0
DS=38EE  ES=38EE  SS=38FE  CS=1FD0  IP=010D   NV UP DI PL ZR NA PE NC
1FD0:010D 2EF606090101   TEST       Byte Ptr CS:[0109],01              CS:0109=00
```

The single Trace command has moved execution into the TSR. Note that the Interrupt 14H has changed the value of CS and jumped to location 10DH off the new CS. This location contains the first instruction of the *COMMSCOPE* procedure in the TSR. SYMDEB does not know that a different segment is being executed and must be instructed to use a different map table. Use the Open Symbol Map command, XO, to do this, instructing SYMDEB to set the active map table to BADSCOP!.

```
-XO BADSCOP!  <Enter>
-X?*  <Enter>

CSEG: (0000)
0100 INITIALIZE   0103 OLD_COMM_INT 0107 COUNT         0109 STATUS
010A PORT         010B BUFPNTR      010D COMMSCOPE     018F CONTROL
020A VECTOR_INIT
```

The X?* command shows that the BADSCOP symbols are now the current map. They are not usable, however, because the value of *CSEG*—zero—needs to be changed to the current CS register. To correct this, use the SYMDEB Set Symbol Value command, Z. This command can set any symbol in the current map table to any value; the value can be a number, another symbol, or the contents of a register. In this case, set the value of *CSEG* in BADSCOP! to the current contents of the CS register.

```
-Z CSEG CS  <Enter>
-X*  <Enter>
 38FE TESTCOMM
       390E CSEG
[0000 BADSCOP]
      [1FD0 CSEG]
```

The X* command confirms that BADSCOP! is now the selected symbol map and that the *CSEG* within it has the value 1FD0H. The *CSEG* segment in TESTCOMM is an entirely different entity and still has its correct value, which will be valid when the TSR returns.

With the symbols set, the debugging can begin by tracing the first few instructions. Because *COMMSCOPE* is not currently active, the routine should quickly pass the processing on to the old interrupt handler.

```
-T5  <Enter>
AX=0300  BX=0000  CX=0133  DX=0000  SP=00F6  BP=0000  SI=0000  DI=0000
DS=38EE  ES=38EE  SS=38FE  CS=1FD0  IP=0113   NV UP DI PL ZR NA PE NC
1FD0:0113 7476          JZ  COMMSCOPE+7E (018B)
AX=0300  BX=0000  CX=0133  DX=0000  SP=00F6  BP=0000  SI=0000  DI=0000
DS=38EE  ES=38EE  SS=38FE  CS=1FD0  IP=018B   NV UP DI PL ZR NA PE NC
1FD0:018B FF2E0301      JMP  FAR [0103]                        DS:0103=0000
AX=0300  BX=0000  CX=0133  DX=0000  SP=00F6  BP=0000  SI=0000  DI=0000
DS=38EE  ES=38EE  SS=38FE  CS=0000  IP=0000   NV UP DI PL ZR NA PE NC
0000:0000 381E6715      CMP  [1567],BL                         DS:1567=00
AX=0300  BX=0000  CX=0133  DX=0000  SP=00F6  BP=0000  SI=0000  DI=0000
DS=38EE  ES=38EE  SS=38FE  CS=0000  IP=0004   NV UP DI PL ZR NA PE NC
0000:0004 BC2CE1        MOV  SP,E12C
AX=0300  BX=0000  CX=0133  DX=0000  SP=E12C  BP=0000  SI=0000  DI=0000
DS=38EE  ES=38EE  SS=38FE  CS=0000  IP=0007   NV UP DI PL ZR NA PE NC
0000:0007 2F            DAS
```

STATUS is tested with a mask of 01H at CS:010DH; the test sets the zero flag, indicating that tracing is disabled. The JZ to *COMMSCOPE+7E* (CS:018BH) is taken. At this address is a far jump to the old Interrupt 14H handler at 1567:1375H. The jump is taken and then disaster strikes. Instead of going to the correct address, processing is suddenly at 0000:0000H. Any wild jump is dangerous, but a far jump into low memory is exceptionally so. This explains the system's locking up and requiring a cold boot to recover.

Now that the bug has been caught in the act, it should be a simple matter to determine what went wrong. When the BADSCOP TSR installed itself, it was seen to place the correct offset address at 0103H. Yet whenever the resident portion of the TSR tries to use the value at that address, it finds all zeros. The initialization routine placed the address at the symbol *OLD_COMM_INT* (1FD0:0103H). If that location is examined, the following is found:

```
-DD OLD_COMM_INT L1  <Enter>
1FD0:0103  1567:1375
```

This is the correct address. Why, then, did the programs find zero there? Use the Display Doublewords command to look at the same memory location again, this time using the specific address 0103H rather than a program symbol.

```
-DD 103 L1  <Enter>
38EE:0103  0000:0000
```

The dump of *OLD_COMM_INT* looked at 1FD0:0103H, but the simple dump looked at 38EE:0103H. The explanation is clear when the values of the registers just before the far jump are examined. The CS register contains 1FD0H and the DS register contains 38EEH.

This is the problem—there is a missing CS override on the indirect jump command. When the TSR installed itself, CS and DS were the same because it was a .COM file. When the TSR is entered as the result of an interrupt call, only CS is set; DS remains what it was in the calling program. Without an override, the CPU assumed that the address of the destination of the far call was located at offset 103H from the DS register. This offset, unfortunately, contained zeros, and the program locked up the system.

The problem is now easily corrected. Exit SYMDEB with the Quit command and edit the program source so that the offending line reads

```
OLD_JUMP:
        JMP     CS:OLD_COMM_INT
```

Debugging C programs with SYMDEB

One of SYMDEB's finest features is the ability to debug with source-line data from programs written in Microsoft C, Pascal, and FORTRAN. The actual lines of C or FORTRAN can be included in the debugging display, and the addresses for breakpoints show which line of code the breakpoints are in. Combined with symbolic debugging, these features provide a powerful tool that can significantly reduce debugging time for programs written in a supported language.

The following rather complicated case illustrates SYMDEB at its best. The program BADSCOP from the previous example was not completely debugged. Although the patch to the BADSCOP code at *OLD_JUMP:* did correct the disastrous problem that caused the system to lock up, running the program in a realistic test situation reveals that a subtle problem still remains that might be in either BADSCOP or one of the support programs.

Before we investigate the problem, a quick review of the programs in the COMMSCOP system is in order. At the heart of the system is the Interrupt 14H intercept program COMMSCOP. When executed, this program installs itself as a TSR and intercepts all Interrupt 14H calls. (The incorrect version of the COMMSCOP program is called BADSCOP.) The installed COMMSCOP TSR passes all Interrupt 14H calls on to the real service routine in the ROM BIOS until it is commanded to start tracing. The COMMSCMD routine controls tracing. This control routine can request that COMMSCOP start, stop, or resume tracing for a specific serial port. These commands are facilitated through Interrupt 60H, which is recognized by the COMMSCOP TSR as a command request. When tracing is started, the trace buffer is emptied by zeroing the trace count and setting the buffer pointer to the first buffer location. When tracing is stopped by COMMSCMD's STOP command, a marker is placed in the buffer to indicate the end of a trace segment. Tracing can be resumed with COMMSCMD's RESUME command. Resuming a trace preserves collected data and places new trace data after the marker in the trace buffer. The RESUME command differs from the START command in that the buffer is not emptied.

Now the problem: When the serial data tracing is started with COMMSCMD (*see* Figure 18-5), data is collected normally. When COMMSCMD issues a STOP command and the data is displayed with COMMDUMP (*see* Figure 18-7), the data appears normal. The traced data ends with a stop mark just as it should. However, the RESUME command of

COMMSCMD causes the stop mark to be overwritten with collected data. After this, whenever COMMDUMP displays data an extra byte appears at the end of the data. The problem could be with either BADSCOP or COMMSCMD. SYMDEB has the facilities to debug both the routines at once.

The first step in the debugging process is, as usual, to gather all the listings and design documentation. As a part of this process, the symbol tables needed for SYMDEB must be prepared. The process of preparing a symbol table for BADSCOP has already been explained; however, preparing the SYMDEB input and supporting listings for a C program is slightly more complicated.

First, when the C program is compiled, three switches must be specified. (C switches are case sensitive and must be entered exactly as shown.)

```
C>MSC /Fc /Zd /Od COMMSCMD;  <Enter>
```

The /Zd switch produces an object file containing line-number information that corresponds to the line numbers of the source file. The /Od switch disables optimization that involves complex code rearrangement; localized optimization, peephole optimization, and other simple forms of optimization are still performed. The /Od switch is not required, but code rearrangement can make the resulting object code more difficult to debug.

The /Fc switch invokes a feature of C that is especially important for debugging with SYMDEB: a listing that contains the C source lines and the generated assembler code intermixed. The file is a .COD file; the command line shown above would produce the file COMMSCMD.COD. Figure 18-12 shows the contents of COMMSCMD.COD.

```
;       Static Name Aliases
;
;       $S142_commands  EQU     commands
        TITLE   commscmd
;       NAME    commscmd.C

        .287
_TEXT   SEGMENT  BYTE PUBLIC 'CODE'
_TEXT   ENDS
_DATA   SEGMENT  WORD PUBLIC 'DATA'
_DATA   ENDS
CONST   SEGMENT  WORD PUBLIC 'CONST'
CONST   ENDS
_BSS    SEGMENT  WORD PUBLIC 'BSS'
_BSS    ENDS
DGROUP  GROUP   CONST,  _BSS,   _DATA
        ASSUME  CS: _TEXT, DS: DGROUP, SS: DGROUP, ES: DGROUP
EXTRN   _int86:NEAR
EXTRN   _printf:NEAR
EXTRN   _stricmp:NEAR
EXTRN   _atoi:NEAR
EXTRN   __chkstk:NEAR
_DATA     SEGMENT
```

Figure 18-12. COMMSCMD.COD.

(more)

```
$SG148  DB      'STOP',  00h
$SG151  DB      'START',  00h
$SG154  DB      'RESUME',  00h
$SG157  DB      0ah, 'Communications tracing %s for port COM%1d:',  0ah,  00h
$S142_commands  DB      'STOPPED',  00h
        ORG     $+2
        DB      'STARTED',  00h
        ORG     $+2
        DB      'RESUMED',  00h
        ORG     $+2
_DATA   ENDS
_TEXT   SEGMENT
;¦*** /*************************************************************************
;¦***  *                                                                       *
;¦***  *  COMMSCMD                                                             *
;¦***  *                                                                       *
;¦***  *  This routine controls the COMMSCOP program that has been in-         *
;¦***  *  stalled as a resident routine.  The operation performed is de-       *
;¦***  *  termined by the command line.  The COMMSCMD program is invoked       *
;¦***  *  as follows:                                                          *
;¦***  *                                                                       *
;¦***  *              COMMSCMD [[cmd][ port]]                                  *
;¦***  *                                                                       *
;¦***  *  where cmd is the command to be executed                              *
;¦***  *            STOP   -- stop trace                                       *
;¦***  *            START  -- flush trace buffer and start trace               *
;¦***  *            RESUME -- resume a stopped trace                           *
;¦***  *        port is the COMM port to be traced (1=COM1, 2=COM2, etc.)      *
;¦***  *                                                                       *
;¦***  *  If cmd is omitted, STOP is assumed.  If port is omitted, 1 is        *
;¦***  *  assumed.                                                             *
;¦***  *                                                                       *
;¦***  *************************************************************************/
;¦***
;¦*** #include <stdlib.h>
;¦*** #include <stdio.h>
;¦*** #include <dos.h>
;¦*** #define COMMSCMD 0x60
;¦***
;¦*** main(argc, argv)
;¦*** int argc;
; Line 29
        PUBLIC _main
_main   PROC NEAR
        *** 000000      55                      push    bp
        *** 000001      8b ec                   mov     bp,sp
        *** 000003      b8 22 00                mov     ax,34
        *** 000006      e8 00 00                call    __chkstk
        *** 000009      57                      push    di
        *** 00000a      56                      push    si
```

Figure 18-12. Continued. *(more)*

```
;*** char *argv[];
;*** {
; Line 31
;       argc = 4
;       argv = 6
;       cmd = -4
;       port = -6
;       result = -2
;       inregs = -34
;       outregs = -20
;***     int cmd, port, result;
;***     static char commands[3] [10] = {"STOPPED", "STARTED", "RESUMED"};
;***     union REGS inregs, outregs;
;***
;***     cmd = 0;
; Line 36
        *** 00000b      c7 46 fc 00 00          mov     WORD PTR [bp-4],0    ;cmd
;***     port = 0;
; Line 37
        *** 000010      c7 46 fa 00 00          mov     WORD PTR [bp-6],0    ;port
;***
;***     if (argc > 1)
; Line 39
        *** 000015      83 7e 04 01             cmp     WORD PTR [bp+4],1    ;argc
        *** 000019      7f 03                   jg      $JCC25
        *** 00001b      e9 5d 00                jmp     $I145
                                        $JCC25:
;***      {
; Line 40
;***      if (0 == stricmp(argv[1], "STOP"))
; Line 41
        *** 00001e      b8 00 00                mov     ax,OFFSET DGROUP:$SG148
        *** 000021      50                      push    ax
        *** 000022      8b 5e 06                mov     bx,[bp+6]            ;argv
        *** 000025      ff 77 02                push    WORD PTR [bx+2]
        *** 000028      e8 00 00                call    _stricmp
        *** 00002b      83 c4 04                add     sp,4
        *** 00002e      3d 00 00                cmp     ax,0
        *** 000031      74 03                   je      $JCC49
        *** 000033      e9 08 00                jmp     $I147
                                        $JCC49:
;***             cmd = 0;
; Line 42
        *** 000036      c7 46 fc 00 00          mov     WORD PTR [bp-4],0    ;cmd
;***         else if (0 == stricmp(argv[1], "START"))
```

Figure 18-12. Continued. *(more)*

```
; Line 43
        *** 00003b      e9 3d 00                jmp     $I149
                                        $I147:
        *** 00003e      b8 05 00                mov     ax,OFFSET DGROUP:$SG151
        *** 000041      50                      push    ax
        *** 000042      8b 5e 06                mov     bx,[bp+6]       ;argv
        *** 000045      ff 77 02                push    WORD PTR [bx+2]
        *** 000048      e8 00 00                call    _stricmp
        *** 00004b      83 c4 04                add     sp,4
        *** 00004e      3d 00 00                cmp     ax,0
        *** 000051      74 03                   je      $JCC81
        *** 000053      e9 08 00                jmp     $I150
                                        $JCC81:
;¦***           cmd = 1;
; Line 44
        *** 000056      c7 46 fc 01 00          mov     WORD PTR [bp-4],1       ;cmd
;¦***        else if (0 == stricmp(argv[1], "RESUME"))
; Line 45
        *** 00005b      e9 1d 00                jmp     $I152
                                        $I150:
        *** 00005e      b8 0b 00                mov     ax,OFFSET DGROUP:$SG154
        *** 000061      50                      push    ax
        *** 000062      8b 5e 06                mov     bx,[bp+6]       ;argv
        *** 000065      ff 77 02                push    WORD PTR [bx+2]
        *** 000068      e8 00 00                call    _stricmp
        *** 00006b      83 c4 04                add     sp,4
        *** 00006e      3d 00 00                cmp     ax,0
        *** 000071      74 03                   je      $JCC113
        *** 000073      e9 05 00                jmp     $I153
                                        $JCC113:
;¦***           cmd = 2;
; Line 46
        *** 000076      c7 46 fc 02 00          mov     WORD PTR [bp-4],2       ;cmd
;¦***        }
; Line 47
                                        $I153:
                                        $I152:
                                        $I149:
;¦***
;¦***    if (argc == 3)
; Line 49
                                        $I145:
        *** 00007b      83 7e 04 03             cmp     WORD PTR [bp+4],3       ;argc
        *** 00007f      74 03                   je      $JCC127
        *** 000081      e9 1b 00                jmp     $I155
                                        $JCC127:
;¦***        {
; Line 50
;¦***        port = atoi(argv[2]);
```

Figure 18-12. Continued.

(more)

```
; Line 51
        *** 000084      8b 5e 06                    mov     bx,[bp+6]               ;argv
        *** 000087      ff 77 04                    push    WORD PTR [bx+4]
        *** 00008a      e8 00 00                    call    _atoi
        *** 00008d      83 c4 02                    add     sp,2
        *** 000090      89 46 fa                    mov     [bp-6],ax               ;port
;¦***          if (port > 0)
; Line 52
        *** 000093      83 7e fa 00                 cmp     WORD PTR [bp-6],0       ;port
        *** 000097      7f 03                       jg      $JCC151
        *** 000099      e9 03 00                    jmp     $I156
                                        $JCC151:
;¦***              port = port-1;
; Line 53
        *** 00009c      ff 4e fa                    dec     WORD PTR [bp-6]         ;port
;¦***          }
; Line 54
                                        $I156:
;¦***
;¦***      inregs.h.ah = cmd;
; Line 56
                                        $I155:
        *** 00009f      8a 46 fc                    mov     al,[bp-4]               ;cmd
        *** 0000a2      88 46 df                    mov     [bp-33],al
;¦***      inregs.x.dx = port;
; Line 57
        *** 0000a5      8b 46 fa                    mov     ax,[bp-6]               ;port
        *** 0000a8      89 46 e4                    mov     [bp-28],ax
;¦***      result = int86(COMMCMD, &inregs, &outregs);
; Line 58
        *** 0000ab      8d 46 ec                    lea     ax,[bp-20]              ;outregs
        *** 0000ae      50                          push    ax
        *** 0000af      8d 46 de                    lea     ax,[bp-34]              ;inregs
        *** 0000b2      50                          push    ax
        *** 0000b3      b8 60 00                    mov     ax,96
        *** 0000b6      50                          push    ax
        *** 0000b7      e8 00 00                    call    _int86
        *** 0000ba      83 c4 06                    add     sp,6
        *** 0000bd      89 46 fe                    mov     [bp-2],ax               ;result
;¦***
;¦***
;¦***      printf("\nCommunications tracing %s for port COM%1d:\n",
;¦***              commands[cmd], port + 1);
; Line 62
        *** 0000c0      8b 46 fa                    mov     ax,[bp-6]               ;port
        *** 0000c3      40                          inc     ax
        *** 0000c4      50                          push    ax
        *** 0000c5      8b 46 fc                    mov     ax,[bp-4]               ;cmd
        *** 0000c8      8b c8                       mov     cx,ax
        *** 0000ca      d1 e0                       shl     ax,1
        *** 0000cc      d1 e0                       shl     ax,1
        *** 0000ce      03 c1                       add     ax,cx
        *** 0000d0      d1 e0                       shl     ax,1
```

Figure 18-12. Continued. *(more)*

```
        *** 0000d2     05 40 00                add     ax,OFFSET DGROUP:$S142_commands
        *** 0000d5     50                      push    ax
        *** 0000d6     b8 12 00                mov     ax,OFFSET DGROUP:$SG157
        *** 0000d9     50                      push    ax
        *** 0000da     e8 00 00                call    _printf
        *** 0000dd     83 c4 06                add     sp,6
;|*** }
; Line 63
                                       $EX138:
        *** 0000e0     5e                      pop     si
        *** 0000e1     5f                      pop     di
        *** 0000e2     8b e5                   mov     sp,bp
        *** 0000e4     5d                      pop     bp
        *** 0000e5     c3                      ret

_main     ENDP
_TEXT     ENDS
END
```

Figure 18-12. Continued.

After the C program is compiled, it must be linked using the /LI switch to indicate that the line number information is to be maintained:

```
C>LINK COMMSCMD /MAP /LI;  <Enter>
```

The /MAP switch is still required to generate a map file of public names for use in building the symbol file, which is created in the usual manner:

```
C>MAPSYM COMMSCMD  <Enter>
```

Everything needed to debug COMMSCMD and BADSCOP is now available. The first test is an attempt to start tracing. To invoke SYMDEB, type

```
C>SYMDEB COMMSCMD.SYM BADSCOP.SYM COMMSCMD.EXE START 1  <Enter>
```

SYMDEB first loads the symbol files for COMMSCMD and BADSCOP and then loads the .EXE file for COMMSCMD. BADSCOP is already in memory, having been loaded by simply running it. (It then stays resident.) The last two entries in the command line load the command tail for COMMSCMD with a start request for COM1. SYMDEB responds with

```
Microsoft (R) Symbolic Debug Utility  Version 4.00
Copyright (C) Microsoft Corp 1984, 1985. All rights reserved.

Processor is [80286]
```

Use the Register and Examine Symbol Map commands to display the initial register values and symbol table information.

```
-R  <Enter>
AX=0000  BX=0000  CX=1928  DX=0000  SP=0800  BP=0000  SI=0000  DI=0000
DS=2CA0  ES=2CA0  SS=2E85  CS=2CB0  IP=010F   NV UP EI PL NZ NA PO NC
_TEXT:__astart:
2CB0:010F B430          MOV  AH,30                      ;'0'
-X*  <Enter>
[2CB0 COMMSCMD]
      [2CB0 _TEXT]
       2E08 DGROUP
 0000 BADSCOP
       0000 CSEG
-X?*  <Enter>
9876 __acrtused    9876 __acrtmsg
_TEXT: (2CB0)
0010 _main         00F6 _atoi
00F9 __chkstk      010F __astart      01AB __cintDIV     01AE __amsg_exit
01B9 _int86        023A _printf       0270 _strcmpi      0270 _stricmp
02C2 __stbuf       0361 __ftbuf       03E7 __catox       043C __nullcheck
0458 __cinit       0507 _exit         051E __exit        054A __ctermsub
0572 __dosret0     057A __dosretax    0586 __maperror    05BA __NMSG_TEXT
05EA __NMSG_WRITE  0613 __output      0E22 __setargv     0F07 __setenvp
0F6D __flsbuf      1098 __fassign     1098 __cropzeros   1098 __positive
1098 __forcdecpt   1098 __cfltcvt     109B _fflush       1103 _isatty
1125 __myalloc     1167 _strlen       1182 _ultoa        118C __fptrap
1192 _flushall     11C3 _free         11C3 __nfree       11D1 _malloc
11D1 __nmalloc     1217 _write        12F1 __cltoasub    12FD __cxtoa
1351 __amalloc     1432 __amexpand    146C __amlink      148E __amallocbrk
14AD _brkctl
DGROUP: (2E08)
0094 STKHQQ        0096 __asizds      0098 __atopsp
009A __abrktb      00EA __abrktbe     00EA __abrkp       00EC __iob
018C __iob2        0204 __lastiob     0212 __aintdiv     0216 __fac
021E _errno        0220 __umaskval    0222 __pspadr      0224 __psp
0226 __osmajor     0226 __dosvermajor 0227 __osminor     0227 __dosverminor
0228 __oserr       0228 __doserrno    022A __osfile      023E ___argc
0240 ___argv       0242 _environ      0244 __child       0246 __csigtab
0278 __cflush      027A __asegds      0286 __aseg1       0288 __asegn
028A __asegr       028C __amblksiz    0292 __fpinit      03A8 _edata
03D0 __bufout      05D0 __bufin       07D0 _end
```

The Register command shows that the first instruction to be executed will be at symbol *__astart* in the _TEXT segment. (Note that C puts a single underscore in front of all public library and routine names; a double underscore indicates routines for C's internal use.) The Examine Symbol Map command reveals that the symbol map COMMSCMD! has two segments, *_TEXT* and *DGROUP*, with *_TEXT* currently selected. The segment in BADSCOP!, *CSEG*, has no value assigned to it because SYMDEB doesn't know where it is; one of the debugging tasks is to determine the location of *CSEG*.

C places initialization and preamble code at the front of its object modules. This code can be skipped during debugging, so this example begins at the label *_main*. Examination of the code at this label using the Disassemble command reveals the following:

```
-U _main  <Enter>
commscmd.C
29:  int argc;
_TEXT:_main:
2CB0:0010 55            PUSH    BP
2CB0:0011 8BEC          MOV     BP,SP
2CB0:0013 B82200        MOV     AX,0022
2CB0:0016 E8E000        CALL    __chkstk
2CB0:0019 57            PUSH    DI
```

This disassembly shows the way source-line information is displayed. These instructions are generated by line 29 of COMMSCMD.C. When the disassembly is compared with the listing in Figure 18-12, the same instructions are seen. However, their addresses are different. The addresses in the disassembly are relative to the start of the segment *_TEXT*, but the addresses in the listing are relative to the start of *_main*. SYMDEB allows address references to be made relative to a symbol, so breakpoints can be set as displacements from *_main* and the addresses shown in the listing can be used.

Because the location of the problem being debugged is not known, breakpoints must be placed strategically throughout COMMSCMD to trace the execution of the program. Use the Set Breakpoints command to set the breakpoints.

```
-BP _main+1e  <Enter>
-BP _main+36  <Enter>
-BP _main+56  <Enter>
-BP _main+76  <Enter>
-BP _main+7b  <Enter>
-BP _main+9c  <Enter>
-BP _main+b7  <Enter>
-BP _main+e5  <Enter>
-BL  <Enter>
0 e 2CB0:002E [_TEXT:_main+1E (002E)] commscmd.C:41
1 e 2CB0:0046 [_TEXT:_main+36 (0046)] commscmd.C:42
2 e 2CB0:0066 [_TEXT:_main+56 (0066)] commscmd.C:44
3 e 2CB0:0086 [_TEXT:_main+76 (0086)] commscmd.C:46
4 e 2CB0:008B [_TEXT:_main+7B (008B)] commscmd.C:49
5 e 2CB0:00AC [_TEXT:_main+9C (00AC)] commscmd.C:53
6 e 2CB0:00C7 [_TEXT:_main+B7 (00C7)] commscmd.C:58
7 e 2CB0:00F5 [_TEXT:_main+E5 (00F5)] commscmd.C:63
```

The List Breakpoints command shows the breakpoint addresses in three ways: first the absolute segment:offset address, then the displacement from the label *_main*, and finally the line number in COMMSCMD.C.

The first part of the COMMSCMD program decodes the arguments and sets the appropriate values for *cmd* and *port*. If there are no arguments, this decoding is skipped; if there are arguments, the decoding begins at line 41, so the first breakpoint is set there. If the criterion of line 41 is met (the first argument is STOP), then line 42 is executed. The second breakpoint is set there. Reaching the second breakpoint means that a STOP command was properly decoded. If the command was not STOP, execution continues at line 43. If this

test is passed, line 44 is executed. This is the location of the third breakpoint. If the test at line 44 fails but the one at line 45 is passed, then the breakpoint at line 46 is executed. Whether or not one of the tests passes, execution ends up at line 49. At this point, the program tests for the presence of a second operand. If there is a second operand, execution traps at line 53, where the program decrements the port number to put it in the proper form for the Interrupt 60H handler. Execution will then always stop in line 58, just before the call to *_int86*. (*_int86* is a library routine that loads registers and executes INT instructions.)

When the program is run with *START1* in the command tail, it gives the following results:

```
-G  <Enter>
AX=0022  BX=0F82  CX=0019  DX=0098  SP=0F7E  BP=0FA4  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=002E   NV UP EI PL NZ NA PO NC
41:               if (0 == stricmp(argv[1],"STOP"))
2CB0:002E B83600          MOV        AX,0036                      ;BR0
-G  <Enter>
AX=0000  BX=415A  CX=0000  DX=0098  SP=0F7E  BP=0FA4  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=0066   NV UP EI PL ZR NA PE NC
44:                       cmd = 1;
2CB0:0066 C746FC0100      MOV        Word Ptr [BP-04],0001        ;BR2 SS:0FA0=0000
-G  <Enter>
AX=0000  BX=415A  CX=0000  DX=0098  SP=0F7E  BP=0FA4  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=008B   NV UP EI PL ZR NA PE NC
49:            if (argc == 3)
2CB0:008B 837E0403        CMP       Word Ptr [BP+04],+03          ;BR4 SS:0FA8=0003
-G  <Enter>
AX=0001  BX=00D0  CX=0000  DX=0000  SP=0F7E  BP=0FA4  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=00AC   NV UP EI PL NZ NA PO NC
5              port = port-1;
2CB0:00AC FF4EFA          DEC        Word Ptr [BP-06]             ;BR5 SS:0F9E=0001
-G  <Enter>
AX=0060  BX=00D0  CX=0000  DX=0000  SP=0F78  BP=0FA4  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=00C7   NV UP EI PL ZR NA PE NC
2CB0:00C7 E8EF00          CALL      _int86                       ;BR6
```

The first break occurs at line 41, indicating that one or more arguments were present in the command line. The next break is at line 44, where the program sets the *cmd* code for Interrupt 60H to 1, the correct value for a start request. The next break occurs at line 49, where the program checks the number of arguments. If this number is 3, then there is a second argument in the command line. (Remember that, in C, the first argument is the name of the routine, so an argument count of 3 actually means that there are 2 arguments present.) The number of arguments is at BP+04, or SS:0FA8H, and it is indeed 3. Therefore, the next break is at line 53. The program decrements the current value of *port*, leaving a value of 0, which is what Interrupt 60H expects to see for COM1.

Continuing execution causes a break just before the call to *_int86*. To validate that the Interrupt 60H call is being made correctly, set a breakpoint just before the INT 60H instruction is issued. Unfortunately, no listing of *_int86* is available, so no alternative

exists but to trace the execution of the routine until the INT instruction is issued. The details of the processing are of no interest to this debugging session, so they can be ignored until an INT 60H is seen. (The trace offers a great deal of information about how C interfaces with subroutines. Studying the trace would be educational but is beyond the scope of this example.)

```
-T 5  <Enter>
AX=0060  BX=00D0  CX=0000  DX=0000  SP=0F76  BP=0FA4  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=01B9   NV UP EI PL ZR NA PE NC
_TEXT:_int86:
2CB0:01B9 55              PUSH     BP
AX=0060  BX=00D0  CX=0000  DX=0000  SP=0F74  BP=0FA4  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=01BA   NV UP EI PL ZR NA PE NC
2CB0:01BA 8BEC            MOV      BP,SP
AX=0060  BX=00D0  CX=0000  DX=0000  SP=0F74  BP=0F74  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=01BC   NV UP EI PL ZR NA PE NC
2CB0:01BC 56              PUSH     SI
AX=0060  BX=00D0  CX=0000  DX=0000  SP=0F72  BP=0F74  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=01BD   NV UP EI PL ZR NA PE NC
2CB0:01BD 57              PUSH     DI
AX=0060  BX=00D0  CX=0000  DX=0000  SP=0F70  BP=0F74  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=01BE   NV UP EI PL ZR NA PE NC
2CB0:01BE 83EC0A          SUB      SP,+0A
-T 5  <Enter>
AX=0060  BX=00D0  CX=0000  DX=0000  SP=0F66  BP=0F74  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=01C1   NV UP EI PL NZ AC PE NC
2CB0:01C1 C646F6CD        MOV      Byte Ptr [BP-0A],CD              SS:0F6A=BE
AX=0060  BX=00D0  CX=0000  DX=0000  SP=0F66  BP=0F74  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=01C5   NV UP EI PL NZ AC PE NC
2CB0:01C5 8B4604          MOV      AX,[BP+04]                       SS:0F78=0060
AX=0060  BX=00D0  CX=0000  DX=0000  SP=0F66  BP=0F74  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=01C8   NV UP EI PL NZ AC PE NC
2CB0:01C8 8846F7          MOV      [BP-09],AL                       SS:0F6B=01
AX=0060  BX=00D0  CX=0000  DX=0000  SP=0F66  BP=0F74  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=01CB   NV UP EI PL NZ AC PE NC
2CB0:01CB 3C25            CMP      AL,25                            ;'%'
AX=0060  BX=00D0  CX=0000  DX=0000  SP=0F66  BP=0F74  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=01CD   NV UP EI PL NZ AC PO NC
2CB0:01CD 740A            JZ       _int86+20 (01D9)
-T 5  <Enter>
AX=0060  BX=00D0  CX=0000  DX=0000  SP=0F66  BP=0F74  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=01CF   NV UP EI PL NZ AC PO NC
2CB0:01CF 3C26            CMP      AL,26                            ;'&'
AX=0060  BX=00D0  CX=0000  DX=0000  SP=0F66  BP=0F74  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=01D1   NV UP EI PL NZ AC PE NC
2CB0:01D1 7406            JZ       _int86+20 (01D9)
AX=0060  BX=00D0  CX=0000  DX=0000  SP=0F66  BP=0F74  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=01D3   NV UP EI PL NZ AC PE NC
2CB0:01D3 C646F8CB        MOV      Byte Ptr [BP-08],CB              SS:0F6C=B0
AX=0060  BX=00D0  CX=0000  DX=0000  SP=0F66  BP=0F74  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=01D7   NV UP EI PL NZ AC PE NC
```

(more)

```
2CB0:01D7 EB0C          JMP     _int86+2C (01E5)
AX=0060  BX=00D0  CX=0000  DX=0000  SP=0F66  BP=0F74  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=01E5   NV UP EI PL NZ AC PE NC
2CB0:01E5 8C56F4        MOV     [BP-0C],SS                    SS:0F68=0F74
-T 5  <Enter>
AX=0060  BX=00D0  CX=0000  DX=0000  SP=0F66  BP=0F74  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=01E8   NV UP EI PL NZ AC PE NC
2CB0:01E8 8D46F6        LEA     AX,[BP-0A]                    SS:0F6A=60CD
AX=0F6A  BX=00D0  CX=0000  DX=0000  SP=0F66  BP=0F74  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=01EB   NV UP EI PL NZ AC PE NC
2CB0:01EB 8946F2        MOV     [BP-0E],AX                    SS:0F66=0060
AX=0F6A  BX=00D0  CX=0000  DX=0000  SP=0F66  BP=0F74  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=01EE   NV UP EI PL NZ AC PE NC
2CB0:01EE 8B7E06        MOV     DI,[BP+06]                    SS:0F7A=0F82
AX=0F6A  BX=00D0  CX=0000  DX=0000  SP=0F66  BP=0F74  SI=0089  DI=0F82
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=01F1   NV UP EI PL NZ AC PE NC
2CB0:01F1 8B05          MOV     AX,[DI]                       DS:0F82=0100
AX=0100  BX=00D0  CX=0000  DX=0000  SP=0F66  BP=0F74  SI=0089  DI=0F82
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=01F3   NV UP EI PL NZ AC PE NC
2CB0:01F3 8B5D02        MOV     BX,[DI+02]                    DS:0F84=0000
-T 5  <Enter>
AX=0100  BX=0000  CX=0000  DX=0000  SP=0F66  BP=0F74  SI=0089  DI=0F82
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=01F6   NV UP EI PL NZ AC PE NC
2CB0:01F6 8B4D04        MOV     CX,[DI+04]                    DS:0F86=0000
AX=0100  BX=0000  CX=0000  DX=0000  SP=0F66  BP=0F74  SI=0089  DI=0F82
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=01F9   NV UP EI PL NZ AC PE NC
2CB0:01F9 8B5506        MOV     DX,[DI+06]                    DS:0F88=0000
AX=0100  BX=0000  CX=0000  DX=0000  SP=0F66  BP=0F74  SI=0089  DI=0F82
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=01FC   NV UP EI PL NZ AC PE NC
2CB0:01FC 8B7508        MOV     SI,[DI+08]                    DS:0F8A=0000
AX=0100  BX=0000  CX=0000  DX=0000  SP=0F66  BP=0F74  SI=0000  DI=0F82
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=01FF   NV UP EI PL NZ AC PE NC
2CB0:01FF 8B7D0A        MOV     DI,[DI+0A]                    DS:0F8C=0000
AX=0100  BX=0000  CX=0000  DX=0000  SP=0F66  BP=0F74  SI=0000  DI=0000
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=0202   NV UP EI PL NZ AC PE NC
2CB0:0202 55            PUSH    BP
-T 5  <Enter>
AX=0100  BX=0000  CX=0000  DX=0000  SP=0F64  BP=0F74  SI=0000  DI=0000
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=0203   NV UP EI PL NZ AC PE NC
2CB0:0203 83ED0E        SUB     BP,+0E
AX=0100  BX=0000  CX=0000  DX=0000  SP=0F64  BP=0F66  SI=0000  DI=0000
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=0206   NV UP EI PL NZ AC PE NC
2CB0:0206 FF5E00        CALL    FAR [BP+00]                   SS:0F66=0F6A
AX=0100  BX=0000  CX=0000  DX=0000  SP=0F60  BP=0F66  SI=0000  DI=0000
DS=2E08  ES=2E08  SS=2E08  CS=2E08  IP=0F6A   NV UP EI PL NZ AC PE NC
2E08:0F6A CD60          INT     60
AX=0100  BX=0000  CX=0000  DX=0000  SP=0F5A  BP=0F66  SI=0000  DI=0000
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=0190   NV UP DI PL NZ AC PE NC
1313:0190 80FC00        CMP     AH,00
AX=0100  BX=0000  CX=0000  DX=0000  SP=0F5A  BP=0F66  SI=0000  DI=0000
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=0193   NV UP DI PL NZ NA PO NC
1313:0193 7521          JNZ     01B6
```

When the Interrupt 60H call is encountered at offset 0F6AH, the values passed to it can be checked. AH contains 1 and DX contains 0—the correct values for START COM1.

In order to use the symbols for BADSCOP, use the Open Symbol Map command, XO, to switch to the correct symbol map. Then, because the value of *CSEG* is not defined in the map, use the Set Symbol Value command to set *CSEG* to the current value of CS. (CS was changed to the correct value for BADSCOP when the program executed the INT 60H instruction.)

```
-XO BADSCOP!  <Enter>
-Z CSEG CS    <Enter>
-X?*          <Enter>

CSEG: (1313)
0100 INITIALIZE   0103 OLD_COMM_INT 0107 COUNT        0109 STATUS
010A PORT         010B BUFPNTR      010D COMSCOPE     0190 CONTROL
020A VECTOR_INIT
```

Because the BADSCOP symbols now have meaning, a great deal of trouble can be avoided by setting a breakpoint at *CONTROL*, the entry point for Interrupt 60H, so that it will no longer be necessary to trace the *_int86* routine to find the INT 60H command. Execution will automatically stop when the Interrupt 60H handler is entered.

```
-BP CONTROL  <Enter>
-BL  <Enter>
0 e 2CB0:002E [COMMSCMD!_TEXT:_main+1E (002E)] commscmd.C:41
1 e 2CB0:0046 [COMMSCMD!_TEXT:_main+36 (0046)] commscmd.C:42
2 e 2CB0:0066 [COMMSCMD!_TEXT:_main+56 (0066)] commscmd.C:44
3 e 2CB0:0086 [COMMSCMD!_TEXT:_main+76 (0086)] commscmd.C:46
4 e 2CB0:008B [COMMSCMD!_TEXT:_main+7B (008B)] commscmd.C:49
5 e 2CB0:00AC [COMMSCMD!_TEXT:_main+9C (00AC)] commscmd.C:53
6 e 2CB0:00C7 [COMMSCMD!_TEXT:_main+B7 (00C7)] commscmd.C:58
7 e 2CB0:00F5 [COMMSCMD!_TEXT:_main+E5 (00F5)] commscmd.C:63
8 e 1313:0190 [CSEGS:CONTROL]
```

With the housekeeping tasks done, the business of debugging BADSCOP can begin. The first thing *CONTROL* does is check for a stop request. If no stop request is present, the routine jumps to the check for a start request. (The first test and jump were already complete when the trace ended above.) The test for a start request is passed. *CONTROL* places the port number in a local variable, resets the buffer pointer and the buffer count, and turns tracing status on. With all this complete, *CONTROL* returns.

```
-T 5  <Enter>
AX=01BB  BX=E81E  CX=3F48  DX=0000  SP=0F5A  BP=0F66  SI=1CE7  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=01B6   NV UP DI PL NZ NA PO NC
1313:01B6 80FC01          CMP     AH,01
AX=01BB  BX=E81E  CX=3F48  DX=0000  SP=0F5A  BP=0F66  SI=1CE7  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=01B9   NV UP DI PL ZR NA PE NC
1313:01B9 751C            JNZ     CONTROL+47 (01D7)
AX=01BB  BX=E81E  CX=3F48  DX=0000  SP=0F5A  BP=0F66  SI=1CE7  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=01BB   NV UP DI PL ZR NA PE NC
1313:01BB 2E88160A01      MOV     CS:[PORT],DL                    CS:010A=00
```

(more)

```
AX=01BB  BX=E81E  CX=3F48  DX=0000  SP=0F5A  BP=0F66  SI=1CE7  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=01C0   NV UP DI PL ZR NA PE NC
1313:01C0 2EC7060B010202 MOV     Word Ptr CS:[BUFPNTR],VECTOR_INIT (0209) CS:010B=0202
AX=01BB  BX=E81E  CX=3F48  DX=0000  SP=0F5A  BP=0F66  SI=1CE7  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=01C7   NV UP DI PL ZR NA PE NC
1313:01C7 2EC70607010000 MOV     Word Ptr CS:[COUNT],0000          CS:0107=0002
-T 5  <Enter>
AX=01BB  BX=E81E  CX=3F48  DX=0000  SP=0F5A  BP=0F66  SI=1CE7  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=01CE   NV UP DI PL ZR NA PE NC
1313:01CE 2EC606090101   MOV     Byte Ptr CS:[STATUS],01           CS:0109=01
AX=01BB  BX=E81E  CX=3F48  DX=0000  SP=0F5A  BP=0F66  SI=1CE7  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=01D4   NV UP DI PL ZR NA PE NC
1313:01D4 EB2B           JMP     CONTROL+71 (0201)
AX=01BB  BX=E81E  CX=3F48  DX=0000  SP=0F5A  BP=0F66  SI=1CE7  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=0201   NV UP DI PL ZR NA PE NC
1313:0201 CF             IRET
AX=01BB  BX=E81E  CX=3F48  DX=0000  SP=0F60  BP=0F66  SI=1CE7  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=2E08  IP=0F6C   NV UP EI PL NZ AC PE NC
2E08:0F6C CB             RETF
AX=01BB  BX=E81E  CX=3F48  DX=0000  SP=0F64  BP=0F66  SI=1CE7  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=0209   NV UP EI PL NZ AC PE NC
2CB0:0209 5D             POP     BP
```

As can be seen from the trace, *CONTROL* performed correctly, so execution of the routine can continue.

```
-G  <Enter>
Communications tracing STARTED for port COM1:
AX=002F  BX=0001  CX=0C13  DX=0000  SP=0FA6  BP=0000  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=00F5   NV UP EI PL NZ NA PE NC
2CB0:00F5 C3             RET                                ;BR7
```

COMMSCMD has written the message to the user and trapped at the breakpoint set at the end of *_main*. The Examine Symbol Map command now shows that SYMDEB has automatically switched to the symbol map for COMMSCMD.

```
-X*  <Enter>
[2CB0 COMMSCMD]
      [2CB0 _TEXT]
       2E08 DGROUP
 0000 BADSCOP
       1313 CSEG
```

No problems have been encountered with the START command; now the same process of checking COMMSCMD and BADSCOP must be repeated for the STOP command. (Even if problems had been found with the START command, it would be imprudent not to test the other commands — they could have errors, too.) SYMDEB could be exited and restarted with new commands, but this would mean the loss of the painfully created set of breakpoints. Instead, a new copy of COMMSCMD is loaded without leaving SYMDEB. One problem with this, however, is that when SYMDEB loads an .EXE file, it adds the value of the initial CS register to the addresses of the segments in the symbol map whose name

matches the .EXE file. This is fine the first time the program loads, but the second time, all the values are doubled and therefore incorrect. To avoid this error, the addresses must be adjusted before the load. Use the Set Symbol Value command to subtract CS from each segment name in COMMSCMD!. The Examine Symbol Map command shows the new values.

```
-Z _TEXT _TEXT-CS  <Enter>
-Z DGROUP DGROUP-CS  <Enter>
-X*  <Enter>
[2CB0 COMMSCMD]
      [0000 _TEXT]
       0158 DGROUP
 0000 BADSCOP
       1313 CSEG
```

The Name File or Command-Tail Parameters command, N, and the Load File or Sectors command, L, can now be used to load a new copy of COMMSCMD.EXE.

```
-N COMMSCMD.EXE  <Enter>
-L  <Enter>
-X*  <Enter>
[2CB0 COMMSCMD]
      [2CB0 _TEXT]
       2E08 DGROUP
 0000 BADSCOP
       1313 CSEG
```

Notice that the segment values inside COMMSCMD! are the same as they were when the program was first loaded. Use the Name command again, this time to set the command tail to contain a STOP command for COM1. The breakpoint table from the first execution is still set, so the program can now be traced in the same way.

```
-N STOP 1  <Enter>
-G  <Enter>
AX=0022  BX=0F84  CX=0019  DX=0098  SP=0F80  BP=0FA6  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=002E   NV UP EI PL NZ NA PO NC
41:             if (0 == stricmp(argv[1],"STOP"))
2CB0:002E B83600          MOV     AX,0036                       ;BR0
-G  <Enter>
AX=0000  BX=415A  CX=0000  DX=0098  SP=0F80  BP=0FA6  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=0046   NV UP EI PL ZR NA PE NC
42:                 cmd = 0;
2CB0:0046 C746FC0000      MOV     Word Ptr [BP-04],0000         ;BR1 SS:0FA2=0000
-G  <Enter>
AX=0000  BX=415A  CX=0000  DX=0098  SP=0F80  BP=0FA6  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=008B   NV UP EI PL ZR NA PE NC
49:         if (argc == 3)
2CB0:008B 837E0403        CMP     Word Ptr [BP+04],+03          ;BR4 SS:0FAA=0003
-G  <Enter>
AX=0001  BX=00D0  CX=0000  DX=0000  SP=0F80  BP=0FA6  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=00AC   NV UP EI PL NZ NA PO NC
53:                 port = port-1;
```

(more)

```
2CB0:00AC FF4EFA          DEC     Word Ptr [BP-06]              ;BR5 SS:0FA0=0001
-G  <Enter>
AX=0060  BX=00D0  CX=0000  DX=0000  SP=0F7A  BP=0FA6  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=00C7   NV UP EI PL ZR NA PE NC
2CB0:00C7 E8EF00          CALL    _int86                        ;BR6
```

COMMSCMD detected that this is a stop request for COM1 and set the arguments for *_int86* correctly. Because a breakpoint is now set at *CONTROL*, tracing until the Interrupt 60H call is found is not necessary. Simply executing the program will cause it to stop at *CONTROL*.

```
-G  <Enter>
AX=001E  BX=3F48  CX=0000  DX=0000  SP=0F5C  BP=0F68  SI=7400  DI=E903
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=0190   NV UP DI PL NZ AC PO NC
CSEG:CONTROL:
1313:0190 80FC00          CMP     AH,00                         ;BR8
```

The registers are set correctly for a stop request on COM1 (AH = 0, DX = 0). The routine can now be traced to check for correct operation. First, however, a quick look at the symbol maps shows that SYMDEB has automatically switched to BADSCOP's symbols.

```
-X*  <Enter>
 2CB0 COMMSCMD
       2CB0 _TEXT
       2E08 DGROUP
[0000 BADSCOP]
      [1313 CSEG]
-T 5  <Enter>
AX=001E  BX=3F48  CX=0000  DX=0000  SP=0F5C  BP=0F68  SI=7400  DI=E903
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=0193   NV UP DI PL ZR NA PE NC
1313:0193 7521            JNZ     CONTROL+26 (01B6)
AX=001E  BX=3F48  CX=0000  DX=0000  SP=0F5C  BP=0F68  SI=7400  DI=E903
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=0195   NV UP DI PL ZR NA PE NC
1313:0195 1E              PUSH    DS
AX=001E  BX=3F48  CX=0000  DX=0000  SP=0F5A  BP=0F68  SI=7400  DI=E903
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=0196   NV UP DI PL ZR NA PE NC
1313:0196 53              PUSH    BX
AX=001E  BX=3F48  CX=0000  DX=0000  SP=0F58  BP=0F68  SI=7400  DI=E903
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=0197   NV UP DI PL ZR NA PE NC
1313:0197 0E              PUSH    CS
AX=001E  BX=3F48  CX=0000  DX=0000  SP=0F56  BP=0F68  SI=7400  DI=E903
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=0198   NV UP DI PL ZR NA PE NC
1313:0198 1F              POP     DS
-T 5  <Enter>
AX=001E  BX=3F48  CX=0000  DX=0000  SP=0F58  BP=0F68  SI=7400  DI=E903
DS=1313  ES=2E08  SS=2E08  CS=1313  IP=0199   NV UP DI PL ZR NA PE NC
1313:0199 C606090100      MOV     Byte Ptr [STATUS],00          DS:0109=01
AX=001E  BX=3F48  CX=0000  DX=0000  SP=0F58  BP=0F68  SI=7400  DI=E903
DS=1313  ES=2E08  SS=2E08  CS=1313  IP=019E   NV UP DI PL ZR NA PE NC
1313:019E 8B1E0B01        MOV     BX,[BUFPNTR]                  DS:010B=0202
AX=001E  BX=0202  CX=0000  DX=0000  SP=0F58  BP=0F68  SI=7400  DI=E903
DS=1313  ES=2E08  SS=2E08  CS=1313  IP=01A2   NV UP DI PL ZR NA PE NC
```

(more)

```
1313:01A2 C60780         MOV     Byte Ptr [BX],80                  DS:0202=80
AX=001E  BX=0202  CX=0000  DX=0000  SP=0F58  BP=0F68  SI=7400  DI=E903
DS=1313  ES=2E08  SS=2E08  CS=1313  IP=01A5   NV UP DI PL ZR NA PE NC
1313:01A5 C64701FF       MOV     Byte Ptr [BX+01],FF               DS:0203=FF
AX=001E  BX=0202  CX=0000  DX=0000  SP=0F58  BP=0F68  SI=7400  DI=E903
DS=1313  ES=2E08  SS=2E08  CS=1313  IP=01A9   NV UP DI PL ZR NA PE NC
1313:01A9 FF060701       INC     Word Ptr [COUNT]                  DS:0107=0000
-T 5  <Enter>
AX=001E  BX=0202  CX=0000  DX=0000  SP=0F58  BP=0F68  SI=7400  DI=E903
DS=1313  ES=2E08  SS=2E08  CS=1313  IP=01AD   NV UP DI PL NZ NA PO NC
1313:01AD FF060701       INC     Word Ptr [COUNT]                  DS:0107=0001
AX=001E  BX=0202  CX=0000  DX=0000  SP=0F58  BP=0F68  SI=7400  DI=E903
DS=1313  ES=2E08  SS=2E08  CS=1313  IP=01B1   NV UP DI PL NZ NA PO NC
1313:01B1 5B             POP     BX
AX=001E  BX=3F48  CX=0000  DX=0000  SP=0F5A  BP=0F68  SI=7400  DI=E903
DS=1313  ES=2E08  SS=2E08  CS=1313  IP=01B2   NV UP DI PL NZ NA PO NC
1313:01B2 1F             POP     DS
AX=001E  BX=3F48  CX=0000  DX=0000  SP=0F5C  BP=0F68  SI=7400  DI=E903
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=01B3   NV UP DI PL NZ NA PO NC
1313:01B3 EB4C           JMP     CONTROL+71 (0201)
AX=001E  BX=3F48  CX=0000  DX=0000  SP=0F5C  BP=0F68  SI=7400  DI=E903
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=0201   NV UP DI PL NZ NA PO NC
1313:0201 CF             IRET
```

CONTROL correctly detected that this was a stop request. It then saved the user's registers and established a DS equal to CS. (Remember that BADSCOP is a .COM file and CS = DS = SS.) Having done this, the routine moves a zero to *STATUS*, which turns the trace off. It then moves 80H FFH to the buffer to indicate the end of a trace session, increments *COUNT* to allow for the new entry, and restores the user's registers. What it does *not* do is increment the buffer pointer to allow for the stop marker. This behavior is entirely consistent with the observed phenomena: When a trace is stopped and resumed, the stop marker is missing and the count is one too high. The fix is to add

```
INC     BX          ;INCREMENT BUFFER POINTER
INC     BX          ; .
MOV     BUFPNTR,BX  ; .
```

to the *CONTROL* procedure before the registers are restored. (Insert these lines later with your favorite editor.)

Even though the bug has been found, the rest of the routine should be checked for other possible bugs.

```
-G  <Enter>
Communications tracing STOPPED for port COM1:
AX=002F  BX=0001  CX=0C13  DX=0000  SP=0FA8  BP=0000  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=00F5   NV UP EI PL NZ AC PO NC
2CB0:00F5 C3             RET                                  ;BR7
```

Loading a new copy of COMMSCMD, setting the command tail to *RESUME 1*, and monitoring program execution yields the following:

```
-N COMMSCMD.EXE  <Enter>
-Z _TEXT _TEXT-CS  <Enter>
-Z DGROUP DGROUP-CS  <Enter>
-X*  <Enter>
[2CB0 COMMSCMD]
      [0000 _TEXT]
       0158 DGROUP
 0000 BADSCOP
       1313 CSEG
-L  <Enter>
-X*  <Enter>
[2CB0 COMMSCMD]
      [2CB0 _TEXT]
       2E08 DGROUP
 0000 BADSCOP
       1313 CSEG
-N RESUME 1  <Enter>
-G  <Enter>
AX=0022  BX=0F82  CX=0019  DX=0098  SP=0F7E  BP=0FA4  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=002E   NV UP EI PL NZ NA PO NC
41:             if (0 == stricmp(argv[1],"STOP"))
2CB0:002E B83600         MOV     AX,0036                        ;BR0
-G  <Enter>
AX=0000  BX=415A  CX=0000  DX=0098  SP=0F7E  BP=0FA4  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=0086   NV UP EI PL ZR NA PE NC
46:                 cmd = 2;
2CB0:0086 C746FC0200     MOV     Word Ptr [BP-04],0002          ;BR3 SS:0FA0=0000
-G  <Enter>
AX=0000  BX=415A  CX=0000  DX=0098  SP=0F7E  BP=0FA4  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=008B   NV UP EI PL ZR NA PE NC
49:         if (argc == 3)
2CB0:008B 837E0403       CMP     Word Ptr [BP+04],+03           ;BR4 SS:0FA8=0003
-G  <Enter>
AX=0001  BX=00D0  CX=0000  DX=0000  SP=0F7E  BP=0FA4  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=00AC   NV UP EI PL NZ NA PO NC
53:                 port = port-1;
2CB0:00AC FF4EFA         DEC     Word Ptr [BP-06]               ;BR5 SS:0F9E=0001
-G  <Enter>
AX=0060  BX=00D0  CX=0000  DX=0000  SP=0F78  BP=0FA4  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=00C7   NV UP EI PL ZR NA PE NC
2CB0:00C7 E8EF00         CALL     _int86                         ;BR6
-G  <Enter>
AX=0265  BX=001E  CX=3F48  DX=0000  SP=0F5A  BP=0F66  SI=0000  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=0190   NV UP DI PL NZ AC PE NC
CSEG:CONTROL:
1313:0190 80FC00         CMP     AH,00                          ;BR8
-T 5  <Enter>
AX=0265  BX=001E  CX=3F48  DX=0000  SP=0F5A  BP=0F66  SI=0000  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=0193   NV UP DI PL NZ NA PO NC
1313:0193 7521           JNZ     CONTROL+26 (01B6)
AX=0265  BX=001E  CX=3F48  DX=0000  SP=0F5A  BP=0F66  SI=0000  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=01B6   NV UP DI PL NZ NA PO NC
1313:01B6 80FC01         CMP     AH,01
```

(more)

```
AX=0265  BX=001E  CX=3F48  DX=0000  SP=0F5A  BP=0F66  SI=0000  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=01B9   NV UP DI PL NZ NA PO NC
1313:01B9 751C          JNZ     CONTROL+47 (01D7)
AX=0265  BX=001E  CX=3F48  DX=0000  SP=0F5A  BP=0F66  SI=0000  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=01D7   NV UP DI PL NZ NA PO NC
1313:01D7 80FC02        CMP     AH,02
AX=0265  BX=001E  CX=3F48  DX=0000  SP=0F5A  BP=0F66  SI=0000  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=01DA   NV UP DI PL ZR NA PE NC
1313:01DA 7516          JNZ     CONTROL+62 (01F2)
-T 5  <Enter>
AX=0265  BX=001E  CX=3F48  DX=0000  SP=0F5A  BP=0F66  SI=0000  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=01DC   NV UP DI PL ZR NA PE NC
1313:01DC 2E833E0B0100   CMP     Word Ptr CS:[BUFPNTR],+00          CS:010B=0202
AX=0265  BX=001E  CX=3F48  DX=0000  SP=0F5A  BP=0F66  SI=0000  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=01E2   NV UP DI PL NZ NA PO NC
1313:01E2 741D          JZ      CONTROL+71 (0201)
AX=0265  BX=001E  CX=3F48  DX=0000  SP=0F5A  BP=0F66  SI=0000  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=01E4   NV UP DI PL NZ NA PO NC
1313:01E4 2E88160A01     MOV     CS:[PORT],DL                       CS:010A=00
AX=0265  BX=001E  CX=3F48  DX=0000  SP=0F5A  BP=0F66  SI=0000  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=01E9   NV UP DI PL NZ NA PO NC
1313:01E9 2EC606090101   MOV     Byte Ptr CS:[STATUS],01            CS:0109=00
AX=0265  BX=001E  CX=3F48  DX=0000  SP=0F5A  BP=0F66  SI=0000  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=01EF   NV UP DI PL NZ NA PO NC
1313:01EF EB10          JMP     CONTROL+71 (0201)
-T 5  <Enter>
AX=0265  BX=001E  CX=3F48  DX=0000  SP=0F5A  BP=0F66  SI=0000  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=0201   NV UP DI PL NZ NA PO NC
1313:0201 CF            IRET
AX=0265  BX=001E  CX=3F48  DX=0000  SP=0F60  BP=0F66  SI=0000  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=2E08  IP=0F6C   NV UP EI PL NZ AC PE NC
2E08:0F6C CB            RETF
AX=0265  BX=001E  CX=3F48  DX=0000  SP=0F64  BP=0F66  SI=0000  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=0209   NV UP EI PL NZ AC PE NC
2CB0:0209 5D            POP     BP
AX=0265  BX=001E  CX=3F48  DX=0000  SP=0F66  BP=0F74  SI=0000  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=020A   NV UP EI PL NZ AC PE NC
2CB0:020A 57            PUSH    DI
AX=0265  BX=001E  CX=3F48  DX=0000  SP=0F64  BP=0F74  SI=0000  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=020B   NV UP EI PL NZ AC PE NC
2CB0:020B 8B7E08        MOV     DI,[BP+08]                         SS:0F7C=0F90
-G  <Enter>
Communications tracing RESUMED for port COM1:
AX=002F  BX=0001  CX=0C13  DX=0000  SP=0FA6  BP=0000  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=00F5   NV UP EI PL NZ NA PE NC
2CB0:00F5 C3            RET                                    ;BR7
-Q  <Enter>
```

The processing of a resume request is correct. Thus, the problem with stop processing in BADSCOP was the only problem. The corrected BADSCOP, which is actually COMMSCOP, is shown in Figure 18-4.

CodeView

CodeView is the most sophisticated debugging monitor produced by Microsoft. It combines the philosophy and many of the commands of its predecessors, DEBUG and SYMDEB, with true source-code debugging. The availability of source lines and symbols allows CodeView to rival the convenience of program development and debugging previously available only in interpreters such as Microsoft GW-BASIC. However, this high level of interaction with the source program is also the root of its problems for advanced debugging.

In order to provide the debugger with the tools to debug at the source-line level and to interrogate program variables, CodeView is required to have a detailed knowledge of how high-order languages work and of their internal conventions. This is not a problem for languages like C, Pascal, and FORTRAN, versions of which are produced by the same company that created CodeView. The object code generated by these compilers obeys a stringent set of rules and conventions. Assembly-language programs, however, tend to follow their own rules and traditions, making them quite different from C programs, with their own separate debugging needs.

C, Pascal, and FORTRAN programmers will find CodeView a dream to use. Assembly-language programmers using versions of MASM earlier than 5.0 will find CodeView cumbersome and will have to weigh its advantages over its disadvantages. All users will, however, appreciate the good design and programming that have gone into CodeView. It is pleasing to know that someone understands the programmer's debugging needs and is trying to ease the burden.

CodeView has added several welcome functions to the debugger's repertoire, but one of these new features towers above the rest — watchpoints. The debugger can watch the values of program variables or expressions and set breakpoints on them, making it possible to stop execution if an expression evaluates to zero or if a location changes. Previous debugging monitors have been limited to tracing and breaking on instructions. This new facet of debugging changes, somewhat, the approach to resolving a bug.

In the previous discussion of debugging techniques, an orderly application of techniques from inspection and observation through instrumentation to debugging monitors was recommended. This sequence is still recommended with CodeView, but now the instrumentation features have been integrated into the debugging monitor.

A simple example

The following example shows how CodeView uses the instrumentation approach to isolate a problem and then uses the debugging monitor functions to solve it. The example is also an introduction to CodeView commands and techniques. The commands are, for the most part, similar to those used by SYMDEB. Those commands that differ greatly are indicated. This example, like all the examples and demonstrations in this article, is not intended to be a complete tutorial — CodeView commands are summarized elsewhere in this book and explained in detail in the manual accompanying the product. *See* PROGRAMMING UTILITIES: CODEVIEW. The example simply shows some of the more common CodeView commands and demonstrates debugging techniques using them.

UPPERCAS.C (Figure 18-13) is a simple program whose sole function is to convert a canned string to uppercase. When executed, the program prints a few of the characters from the string and some that aren't in the string. Inspecting the listing doesn't reveal the cause of the problem. (Some readers with experience writing C programs will see the cause of the problem, because it is quite common; pretend, for now, that the listing is of no help and enjoy the wonders of CodeView.)

```
/*************************************************************************
 *                                                                       *
 * UPPERCAS.C                                                            *
 *    This routine converts a fixed string to uppercase and prints it.   *
 *                                                                       *
 *************************************************************************/

#include <ctype.h>
#include <string.h>
#include <stdio.h>

main(argc,argv)

int argc;
char *argv[];

{
char    *cp,c;

        cp = "a string\n";

        /*  Convert *cp to uppercase and write to standard output  */

        while (*cp != '\0')
                {
                c = toupper(*cp++);
                putchar(c);
                }

}
```

Figure 18-13. An erroneous C program to convert a string to uppercase.

Like SYMDEB, CodeView requires some special preparation to produce a suitable executable file. CodeView, however, makes the job much simpler. Using the Microsoft C Compiler, compile the program with

```
C>MSC /Zi UPPERCAS;  <Enter>
```

(Remember that C is case sensitive when interpreting switches, so the /Zi switch should be entered exactly as shown.) The /Zi switch instructs the compiler to generate the symbol tables and line-number information needed by CodeView. Other options appropriate to the program can also be included, but /Zi is required.

To form an executable file, use the Microsoft Object Linker (LINK) as follows:

```
C>LINK /CO UPPERCAS;  <Enter>
```

This command line instructs LINK to build an executable file with the information needed for CodeView. Other options can be used as needed or desired. The output of LINK, UPPERCAS.EXE, will be larger than a .EXE file built without /CO (about 2600 bytes larger in this case), but the program will run correctly when executed without CodeView.

Starting CodeView is straightforward. Simply type

```
C>CV UPPERCAS  <Enter>
```

CodeView loads UPPERCAS.EXE. It locates UPPERCAS.C, the source file, and loads that too. It then presents a full-screen display similar to this:

```
File  View  Search  Run  Watch  Options  Language  Calls  Help | F8=Trace F5=Go
                              uppercas.C
1:
2:      /*************************************************************************
3:       *
4:       * UPPERCAS.C
5:       *    This routine converts a fixed string to uppercase and prints it.
6:       *
7:       *************************************************************************
8:
9:      #include <ctype.h>
10:     #include <string.h>
11:     #include <stdio.h>
12:
13:     main(argc,argv)
14:
15:     int argc;
16:     char *argv[];
17:
18:     {

Microsoft (R) CodeView (R)  Version 2.0
(C) Copyright Microsoft Corp. 1986, 1987.  All rights reserved.
>
```

This display has two windows open: the display window, which shows the program being debugged, and the the dialog window, which currently contains only the copyright notice and a prompt (>) for input. The F6 function key moves the cursor back and forth between the two windows.

CodeView can be instructed from either window to go to a specific line (that is, to execute until a specific line is reached). If the cursor is in the display window, use the arrow keys to select a line and press the F7 key. Execution will proceed until the selected line (or the end of the program) is reached. To start execution without specifying a stop line, press F5.

The same functions can be performed from the dialog window using typed commands, which may seem more familiar. Enter the Go Execute Program command, G, optionally followed by an address. Execution will continue until the specified address is reached

or until stopped by something else, such as the end of the program. In this sense, the CodeView Go command is the same as that of DEBUG and SYMDEB. Unlike those routines, however, CodeView's Go command does not allow an equals operator (=).

The address for the Go command can be specified in several ways. Because the display window is currently showing only source lines, it is appropriate to set the stop location in terms of line numbers. The syntax of a line-number specification is the same as in SYMDEB — simply enter the line number preceded by a period:

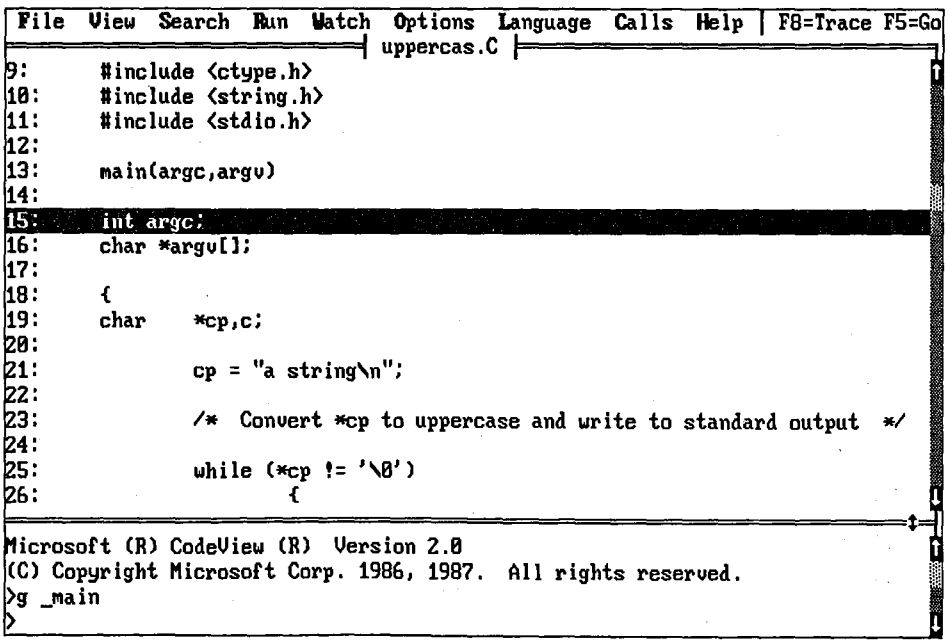
```
>G .27  <Enter>
```

Note that the line number is specified in decimal. This seemingly innocent statement uncovers one of the problem areas in CodeView, especially for assembly-language programmers. The default radix for CodeView is decimal. This convention works well for things associated with the C program, such as line numbers, but is very inconvenient for addresses and other similar items, which are usually in hexadecimal. Hexadecimal numbers must be specified using the cumbersome C notation. Thus, the number FF3EH would be entered as 0xff3e. The radix can be changed using the Change Current Radix command, N (different from the DEBUG and SYMDEB N command). (The problems associated with hexadecimal numbers in early versions of CodeView are no longer present in versions 2.0 and later.)

The radix problem can be avoided, for the moment, by using labels. Issue

```
>G _main  <Enter>
```

to cause CodeView to execute until the main routine is reached. CodeView then shows



The display shows line 15 in reverse video, indicating that CodeView has stopped there. This is the first line of the *main()* module, but it is not executable. Press the F10 key, which has the same effect as entering the Step Through Program command, P, in the dialog window, to cause line 19 to be executed. The reverse video line is then 21, which is the next line to be executed.

To see the changes to *cp*, **cp*, and *c*, establish a watch on these three variables. To use the Watch Word command, WW, for the word *cp*, type

```
>WW cp  <Enter>
```

When entered from the dialog window, this command opens the watch window at the top of the screen and displays the current value of *cp*. To display the expression at **cp*, use the Watch Expression command, W?, as follows:

```
>W? cp,s  <Enter>
```

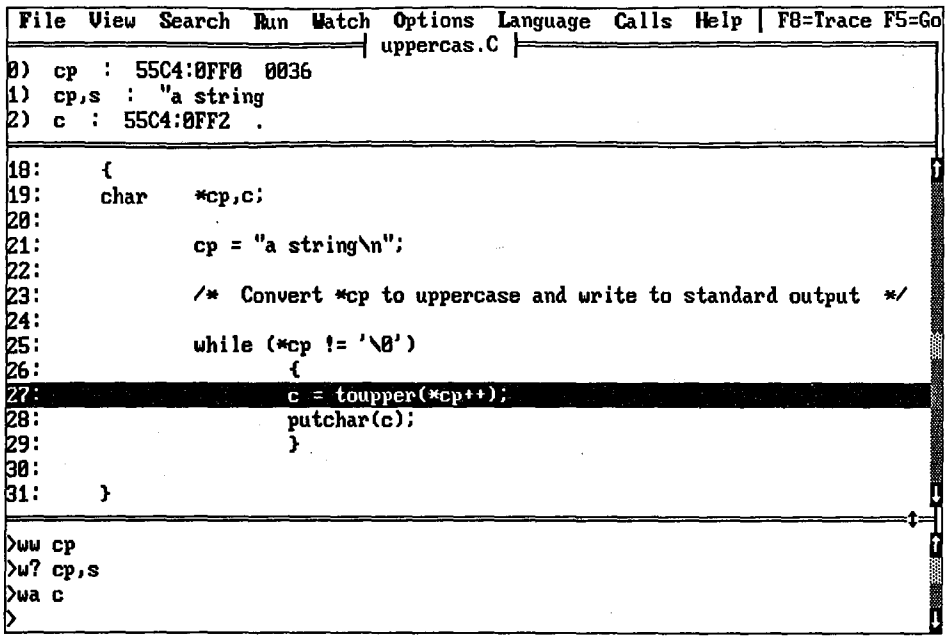
This expression will display the null-delimited string at **cp*. Finally, to see the ASCII character value of *c*, use the Watch ASCII command, WA:

```
>WA c  <Enter>
```

The results of these watch commands are shown in the following screen:

```
File  View  Search  Run  Watch  Options  Language  Calls  Help | F8=Trace F5=Go
                                uppercas.C
0)  cp  :  55C4:0FF0  5527
1)  cp,s  :  ""
2)  c  :  55C4:0FF2  .

9:      #include <ctype.h>
10:     #include <string.h>
11:     #include <stdio.h>
12:
13:     main(argc,argv)
14:
15:     int argc;
16:     char *argv[];
17:
18:     {
19:     char    *cp,c;
20:
21:             cp = "a string\n";
22:

>ww cp
>w? cp,s
>wa c
>
```

The values displayed in the watch window are not yet defined because line 21, which initialized *cp*, has not been executed. Press F8 to rectify this. Press it again to bring the execution of the program into the main loop.



The pointer *cp* now contains the correct address. The Display Memory command, D, could be used to display the contents of DS:0036H, just as in DEBUG and SYMDEB. (This step is not necessary, however, because there is a formatted display of memory in the watch window at 1). The variable *c* has not yet been initialized.

Press the F8 key to execute line 27. A curious and unexpected thing happens, as shown in the next screen:

```
File  View  Search  Run  Watch  Options  Language  Calls  Help | F8=Trace F5=Go
                              uppercas.C
0)  cp  :  55C4:0FF0  0038
1)  cp,s  :  "string
2)  c  :  55C4:0FF2

18:     {
19:     char    *cp,c;
20:
21:             cp = "a string\n";
22:
23:             /*  Convert *cp to uppercase and write to standard output  */
24:
25:             while (*cp != '\0')
26:                     {
27:                     c = toupper(*cp++);
28:                     putchar(c);
29:                     }
30:
31:     }

>ww cp
>w? cp,s
>wa c
>
```

Notice that the value of *cp* has changed from 0036H to 0038H. The line of code, however, indicates that the pointer should have been incremented by only one (**cp++*). The second character of the string, a blank, has been loaded into *c*. This could explain the apparent random selection of characters being displayed (actually every other character) and the garbage characters displayed (the zero at the end of the string might be skipped, causing the routine to continue converting until a zero is encountered somewhere in memory).

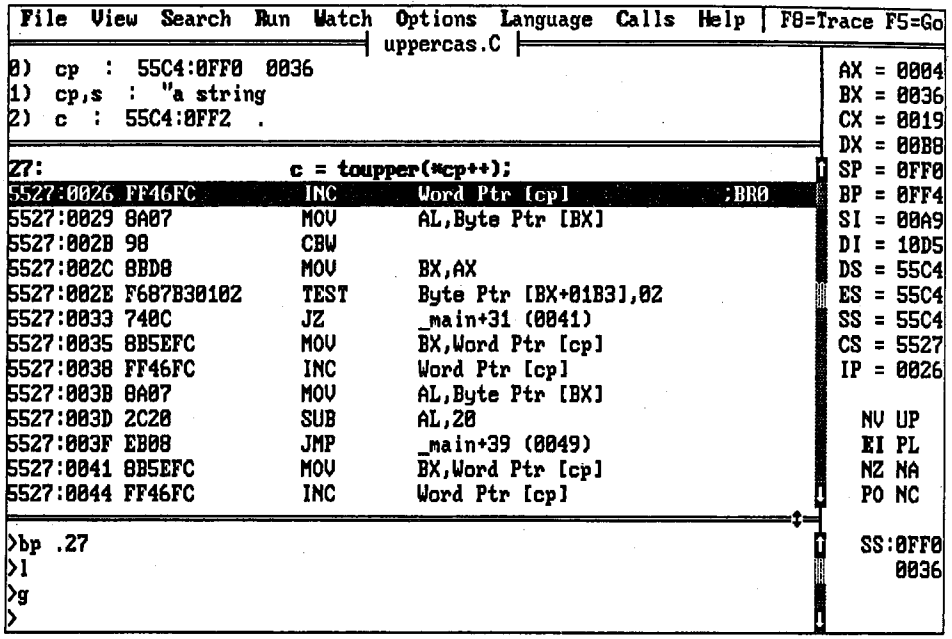
Source-line debugging does not reveal enough about what is happening in this case. To look more closely at the mechanism of the program, the program must be restarted. Before doing this, set a breakpoint at line 27:

```
>BP .27  <Enter>
```

Then restart (actually, reload) the program with the Reload Program command, L. Note that watch commands and breakpoints are preserved when a program is restarted. Executing the restarted program with G yields

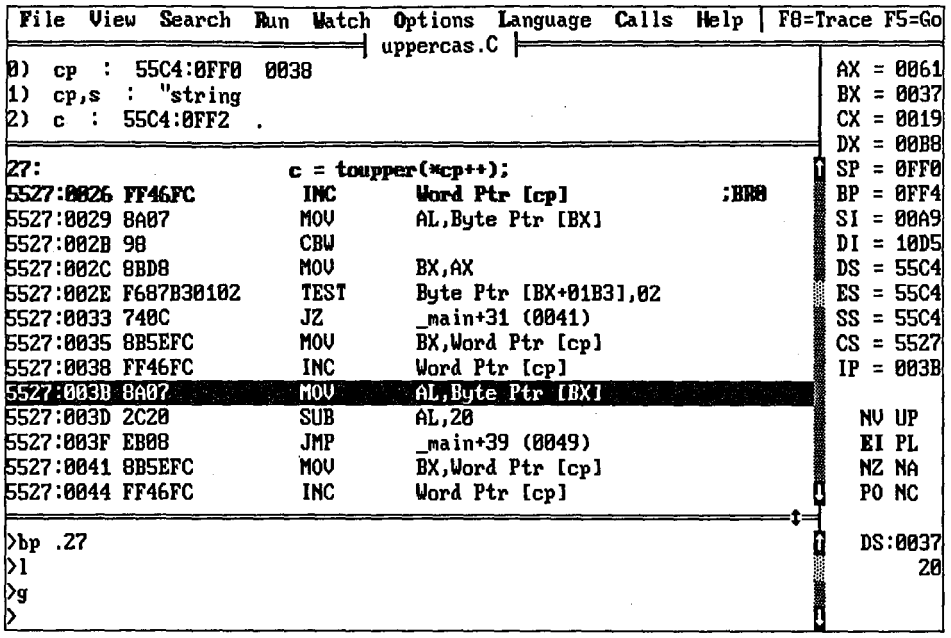
```
File  View  Search  Run  Watch  Options  Language  Calls  Help | F8=Trace F5=Go
                          uppercas.C
0)  cp  :  55C4:0FF0  0036
1)  cp,s  :  "a string
2)  c  :  55C4:0FF2  .

18:     {
19:     char    *cp,c;
20:
21:             cp = "a string\n";
22:
23:             /*  Convert *cp to uppercase and write to standard output  */
24:
25:             while (*cp != '\0')
26:                     {
27:                     c = toupper(*cp++);
28:                     putchar(c);
29:                     }
30:
31:     }

>bp .27
>l
>g
>
```

The display shows line 27 in reverse video, indicating that it is the next line to be executed. The pointer *cp* has the correct value, as shown in the watch window. Now Press the F2 key to turn on the register display and press F3 to show the assembly code.



The display highlights line 27, indicating that a breakpoint exists at this line. The line of code at CS:0026H is in reverse video, indicating that it is the next line to be executed.

The previous instruction has loaded BX with *[cp]*. The first thing the code for line 27 does is increment the word at memory location *[cp]*. The initial value of *cp* is in BX, so the **cp++* request can now be executed. Use the F8 key to single-step through the lines of code. Notice that when only source lines are on the screen, F8 steps one source line at a time, but when assembly code is shown, F8 steps one assembly line at a time. Single-stepping through the code, note how the registers and watch window change. Everything appears normal until CS:0038H is executed.



Notice that the value of *cp* in the watch window has incremented again. The line of C code has two increments hidden in it, not the expected single increment. Why is this?

To find the answer, examine the *toupper()* macro. The following definition, extracted from CTYPE.H, explains what is happening:

```
#define _UPPER          0x1       /* uppercase letter */
#define _LOWER          0x2       /* lowercase letter */
#define isupper(c)       ( (_ctype+1)[c] & _UPPER )
#define islower(c)       ( (_ctype+1)[c] & _LOWER )

#define _tolower(c)      ( (c)-'A'+'a' )
#define _toupper(c)      ( (c)-'a'+'A' )

#define toupper(c)       ( (islower(c)) ? _toupper(c) : (c) )
#define tolower(c)       ( (isupper(c)) ? _tolower(c) : (c) )
```

The argument to *toupper()*, *c*, is used twice, once in the macro that checks for lowercase, *islower()*, and once in *_toupper()*. The argument is replaced in this case with **cp++*, which has the famous C unexpected side effects. Because the unary post-increment is the handiest way to perform the function desired in the program, fixing the problem by changing the code in the main loop is undesirable. Another solution to the problem is to use the function version of *toupper()*. Because *toupper()* is defined as a function in STDIO.H, simply deleting *#include <ctype.h>* would solve the problem. Unfortunately, this would also deprive the program of the other useful definitions in CTYPE.H. (Admittedly, the features are not currently used by the program, but little programs sometimes grow into mighty systems.) So to keep CTYPE.H but still remove the macro definition of

toupper(), use the #undef command. (Because *tolower()* has the same problem, it should also be undefined.) The corrected listing is shown in Figure 18-14.

```
/*************************************************************************
 *                                                                       *
 * UPPERCAS.C                                                            *
 *    This routine converts a fixed string to uppercase and prints it.   *
 *                                                                       *
 *************************************************************************/

#include <ctype.h>
#undef toupper
#undef tolower
#include <string.h>
#include <stdio.h>

main(argc,argv)

int argc;
char *argv[];

{
char    *cp,c;

        cp = "a string\n";

        /*  Convert *cp to uppercase and write to standard output  */

        while (*cp != '\0')
                {
                c = toupper(*cp++);
                putchar(c);
                }

}
```

Figure 18-14. The corrected version of UPPERCAS.C.

An example using screen output

A problem with DEBUG is that it writes to the same screen as the program does. Both SYMDEB and CodeView, however, allow the debugger to switch back and forth between the screen containing the program's output and the screen containing the debugger's output. This feature is a special option with SYMDEB and is sometimes clumsy to use, but with CodeView, keeping a separate program output screen is automatic and switching back and forth involves simply pressing a function key (F4).

The following example program is intended to display an ASCII lookup table with all the displayable characters available on an IBM PC. The expected output is shown in Figure 18-15.

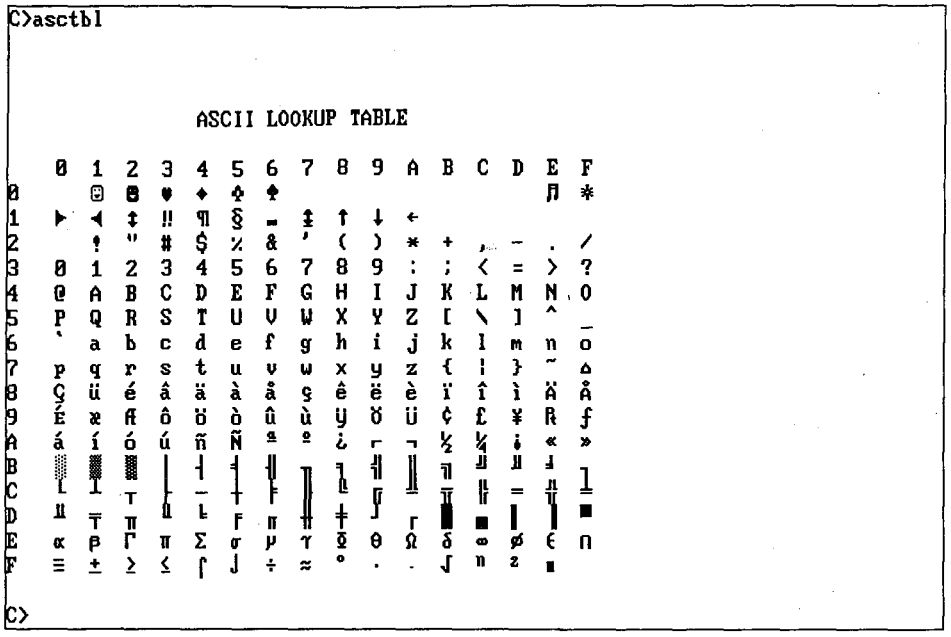


Figure 18-15. The output expected from ASCTBL.C.

The program that should produce this display, ASCTBL.C, is shown in Figure 18-16.

```
/*************************************************************************
 *                                                                       *
 *  ASCTBL.C                                                             *
 *   This program generates an ASCII lookup table for all displayable    *
 *   ASCII and extended IBM PC codes, leaving blanks for nondisplayable  *
 *   codes.                                                              *
 *                                                                       *
 *************************************************************************/

#include <ctype.h>
#include <stdio.h>

main()
{
int i, j, k;
        /* Print table title. */
        printf("\n\n\n                ASCII LOOKUP TABLE\n\n");

        /* Print column headers. */
        printf("    ");
        for (i = 0; i < 16; i++)
                printf("%X  ", i);
        fputchar("\n");
```

Figure 18-16. An erroneous program to display ASCII characters. *(more)*

```
        /* Print each line of the table. */
        for (i = 0, k = 0; i < 16; i++)
                {
                /* Print first hex digit of symbols on this line. */
                printf("%X   ", i);
                /* Print each of the 16 symbols for this line. */
                for (j = 0; j < 16; j++)
                        {
                        /* Filter nonprintable characters. */
                        if ((k >= 7 && k <= 13) || (k >= 28 && k <= 31))
                                printf("   ");
                        else
                                printf("%c  ", k);
                        k++;
                        }
                fputchar("\n");
                }
}
```

Figure 18-16. Continued.

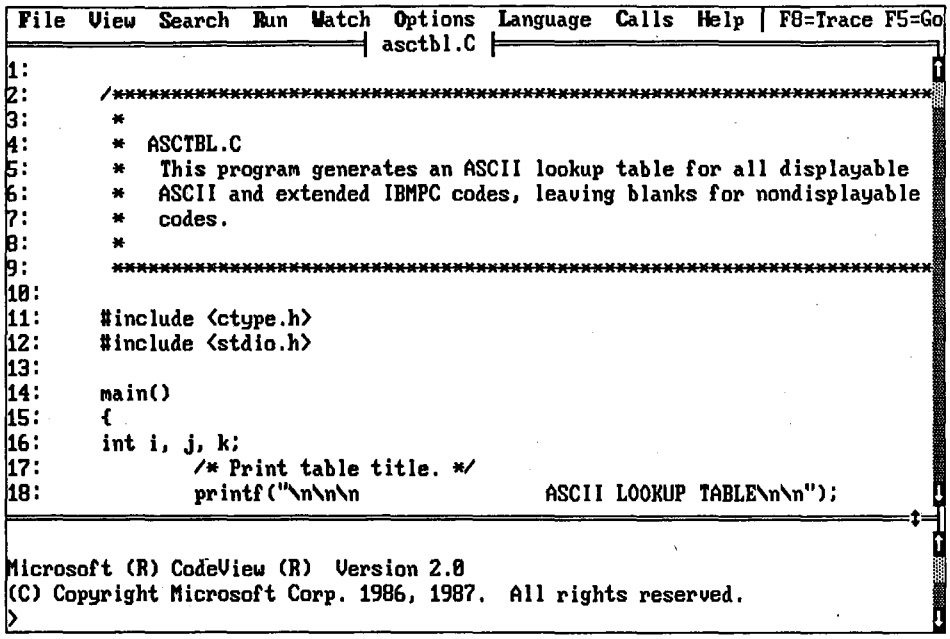
The problem to be debugged in this example is evident when the program in Figure 18-16 is compiled, linked, and executed. Here is the resulting display:

```
C>asctbl


                ASCII LOOKUP TABLE

   0 1 2 3 4 5 6 7 8 9 A B C D E F h0     ☺ ☻ ♥ ♦ ♣ ♠
                  ‼ * y1  ▶ ◀ ↕ !! ¶ § ▬ ↨ ↑ ↓ →           y2
  ! " # $ % & ' ( ) * + , - . / y3  0 1 2 3 4 5 6 7 8
9 : ; < = > ? y4  @ A B C D E F G H I J K L M N O y5  P
 Q R S T U V W X Y Z [ \ ] ^ _ y6  ` a b c d e f g h i
 j k l m n o y7  p q r s t u v w x y z { | } ~ ⌂ y8  Ç
 ü é â ä à å ç ê ë è ï î ì Ä Å y9  É æ Æ ô ö ò û ù ÿ Ö
 Ü ¢ £ ¥ ₧ ƒ yA  á í ó ú ñ Ñ ª º ¿ ⌐ ¬ ½ ¼ ¡ « » yB  ░
▓ ▓ │ ┤ ╡ ╢ ╖ ╕ ╣ ║ ╗ ╝ ╜ ╛ ┐ yC  └ ┴ ┬ ├ ─ ┼ ╞ ╟ ╚ ╔ ╩
╦ ╠ ═ ╬ ╧ ╨ yD  ╤ ╥ ╙ ╘ ╒ ╓ ╫ ╪ ┘ ┌ ▓ ▓ ▌ ▐ ▀ yE  α ß
 Γ π Σ σ µ τ Φ Θ Ω δ ∞ ø ϵ ∩ yF  ≡ ± ≥ ≤ ⌠ ⌡ ÷ ≈ ° ∙ ·
 √ n ² ■   y
C>
```

Something is clearly wrong. The output is jumbled and no pattern is immediately obvious. To locate the problem, first prepare a .EXE file and start CodeView as follows:

```
C>MSC /Zi ASCTBL;  <Enter>
C>LINK /CO ASCTBL;  <Enter>
C>CV ASCTBL  <Enter>
```

CodeView starts and displays the following screen:



The start of the source program is shown in the display window and the dialog window contains an input prompt. Press the F10 key three times to bring execution to line 21. (Remember that the line indicated in reverse video has not yet been executed.)

```
File  View  Search  Run  Watch  Options  Language  Calls  Help | F8=Trace F5=Go
                              asctbl.C
9:      ************************************************************************
10:
11:     #include <ctype.h>
12:     #include <stdio.h>
13:
14:     main()
15:     {
16:     int i, j, k;
17:             /* Print table title. */
18:             printf("\n\n\n                ASCII LOOKUP TABLE\n\n");
19:
20:             /* Print column headers. */
21:             printf("    ");
22:             for (i = 0; i < 16; i++)
23:                     printf("%X  ", i);
24:             fputchar("\n");
25:
26:             /* Print each line of the table. */

Microsoft (R) CodeView (R)  Version 2.0
(C) Copyright Microsoft Corp. 1986, 1987.  All rights reserved.
>
```

The display heading has been printed at line 18. Press the F4 key to display what the program has written on the screen.

```
C>cv asctbl


                ASCII LOOKUP TABLE
```

Note: Any information on the screen when you started CodeView will remain on the virtual output screen until program execution clears it or forces it to scroll off.

The table heading has been properly written to the screen. Press the F4 key again to return to the CodeView display. Continue executing the program with the F10 key to bring the program to line 24.

```
File  View  Search  Run  Watch  Options  Language  Calls  Help | F8=Trace F5=Go
                              asctbl.C
9:      ************************************************************************
10:
11:     #include <ctype.h>
12:     #include <stdio.h>
13:
14:     main()
15:     {
16:     int i, j, k;
17:             /* Print table title. */
18:             printf("\n\n\n                ASCII LOOKUP TABLE\n\n");
19:
20:             /* Print column headers. */
21:             printf("    ");
22:             for (i = 0; i < 16; i++)
23:                     printf("%X  ", i);
24:             fputchar("\n");
25:
26:             /* Print each line of the table. */

Microsoft (R) CodeView (R)  Version 2.0
(C) Copyright Microsoft Corp. 1986, 1987.  All rights reserved.
>
```

At this point in program execution, the column headings have been written on the screen. Press the F4 key again to see the results.

```
C>cv asctbl


              ASCII LOOKUP TABLE

   0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
```

The output of the program is still correct, so allow execution to continue by pressing F4 to return to the CodeView screen and then pressing the F10 key. This will execute the call to the *fputchar()* function to write a newline character.

```
File  View  Search  Run  Watch  Options  Language  Calls  Help | F8=Trace F5=Go
                              asctbl.C
21:             printf("    ");
22:             for (i = 0; i < 16; i++)
23:                     printf("%X  ", i);
24:             fputchar("\n");
25:
26:             /* Print each line of the table. */
27:             for ( i = 0, k = 0; i < 16; i++)
28:                     {
29:                     /* Print first hex digit of symbols on this line. */
30:                     printf("%X   ", i);
31:                     /* Print each of the 16 symbols for this line. */
32:                     for (j = 0; j < 16; j++)
33:                             {
34:                             /* Filter non-printable characters. */
35:                             if ((k >= 7 && k <= 13) || (k >= 28 && k <= 31)
36:                                     printf("   ");
37:                             else
38:                                     printf("%c  ", k);

Microsoft (R) CodeView (R)  Version 2.0
(C) Copyright Microsoft Corp. 1986, 1987.  All rights reserved.
>
```

Examination of the output screen shows that the display is now incorrect.

```
C>cv asctbl

              ASCII LOOKUP TABLE

  0 1 2 3 4 5 6 7 8 9 A B C D E F h
```

A lowercase *h* has been written to the screen instead of a newline character. Further execution demonstrates that newline characters written with *fputchar()* are not working. A closer inspection of the *fputchar()* function is needed.

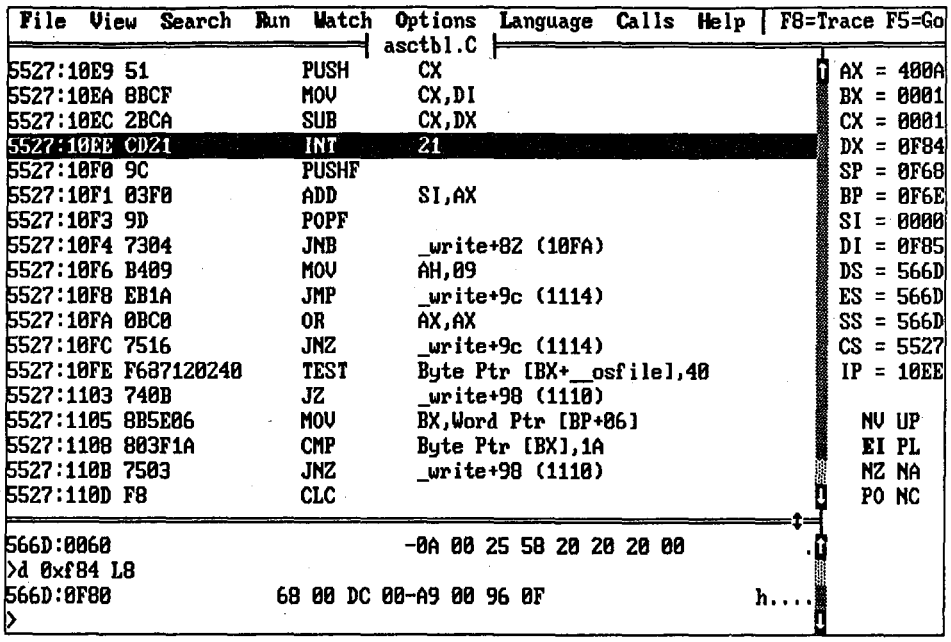
To see what is happening, use the Reload Program command to restart execution at the top of the program. Change the cursor window with the F6 key, use the arrow keys to place the cursor on line 24, and press F7. This brings execution back to line 24, where *fputchar()* is called. Press the F3 key to place the display in assembly mode and the F2 key to show the CPU registers and flags. The first assembly instruction of the *fputchar()* function call is about to be executed.

```
File  View  Search  Run  Watch  Options  Language  Calls  Help | F8=Trace F5=Go
                              asctbl.C
24:             fputchar("\n");                                  AX = 0003
5527:004E B86800        MOV     AX,0068                          BX = 0001
5527:0051 50            PUSH    AX                               CX = 0001
5527:0052 E83F01        CALL    _fputchar (0194)                 DX = 03C0
5527:0055 83C402        ADD     SP,+02                           SP = 0F90
27:             for ( i = 0, k = 0; i < 16; i++)                 BP = 0F96
5527:0058 C746FE0000    MOV     Word Ptr [i],0000                SI = 00A9
5527:005D C746FA0000    MOV     Word Ptr [k],0000                DI = 1075
5527:0062 837EFE10      CMP     Word Ptr [i],+10                 DS = 566D
5527:0066 7D68          JGE     _main+c0 (00D0)                  ES = 566D
5527:0068 EB05          JMP     _main+5f (006F)                  SS = 566D
5527:006A FF46FE        INC     Word Ptr [i]                     CS = 5527
5527:006D EBF3          JMP     _main+52 (0062)                  IP = 004E
30:                     printf("%X  ", i);
5527:006F FF76FE        PUSH    Word Ptr [i]                       NV UP
5527:0072 B86A00        MOV     AX,006A                            EI PL
5527:0075 50            PUSH    AX                                 ZR NA
5527:0076 E84801        CALL    _printf (01C1)                     PE NC

Microsoft (R) CodeView (R)  Version 2.0
(C) Copyright Microsoft Corp. 1986, 1987.  All rights reserved.
>l
>
```

Notice that the parameter being passed to the function by means of the stack is 0068H. Use the Display Memory command to display DS:0068H. (Note the hexadecimal notation.)

```
File  View  Search  Run  Watch  Options  Language  Calls  Help | F8=Trace F5=Go
                              asctbl.C
24:             fputchar("\n");                                  AX = 0003
5527:004E B86800        MOV     AX,0068                          BX = 0001
5527:0051 50            PUSH    AX                               CX = 0001
5527:0052 E83F01        CALL    _fputchar (0194)                 DX = 03C0
5527:0055 83C402        ADD     SP,+02                           SP = 0F90
27:             for ( i = 0, k = 0; i < 16; i++)                 BP = 0F96
5527:0058 C746FE0000    MOV     Word Ptr [i],0000                SI = 00A9
5527:005D C746FA0000    MOV     Word Ptr [k],0000                DI = 1075
5527:0062 837EFE10      CMP     Word Ptr [i],+10                 DS = 566D
5527:0066 7D68          JGE     _main+c0 (00D0)                  ES = 566D
5527:0068 EB05          JMP     _main+5f (006F)                  SS = 566D
5527:006A FF46FE        INC     Word Ptr [i]                     CS = 5527
5527:006D EBF3          JMP     _main+52 (0062)                  IP = 004E
30:                     printf("%X  ", i);
5527:006F FF76FE        PUSH    Word Ptr [i]                       NV UP
5527:0072 B86A00        MOV     AX,006A                            EI PL
5527:0075 50            PUSH    AX                                 ZR NA
5527:0076 E84801        CALL    _printf (01C1)                     PE NC

>l
>d 0x68 L8
566D:0060                       -0A 00 25 58 20 20 20 00        .
>
```

The contents of memory at this address consist of a null-delimited string containing a newline character. The representation of \n is correct. To see how the string is handled, use the trace key, F8, to single-step through *fputchar()* and subordinate functions. These functions are complicated; nearly 100 steps are required to reach the MS-DOS Interrupt 21H call that actually writes the screen.



The AH register's contents, 40H, indicate that the Interrupt 21H call is a request for a write to a device. The BX register has the handle of the device, 1, which is the special file handle for standard output (*stdout*). For this program as it was invoked, standard output is the screen. The CX register indicates that 1 byte is to be written; DS:DX points to the data to be written. The contents of memory at DS:0F84H finally reveal the cause of the problem: This memory location contains the *address* of the data to be written, not the data. The *fputchar()* function was called with the wrong level of indirection.

Examination of the listing shows that all the newline requests were made with

```
fputchar("\n");
```

Strings specified with double quotes are replaced in C functions with the address of the string, but the function expected the actual character and not its address. The problem can be corrected by replacing the *fputchar()* calls with

```
fputchar('\n');
```

The newline character will now be passed directly to the function.

This kind of problem can be avoided. C provides the ability to check the type of each parameter passed to a function against the expected type. If the following definition is included at the top of the C program, incorrect types will generate error messages:

```
#define LINT_ARGS
```

The corrected listing is shown in Figure 18-17. This new program produces the correct output.

```
/*************************************************************************
 *                                                                       *
 *  ASCTBL.C                                                             *
 *   This program generates an ASCII lookup table for all displayable    *
 *   ASCII and extended IBM PC codes, leaving blanks for nondisplayable  *
 *   codes.                                                              *
 *                                                                       *
 *************************************************************************/

#define LINT_ARGS
#include <ctype.h>
#include <stdio.h>

main()
{
int i, j, k;
        /* Print table title. */
        printf("\n\n\n                ASCII LOOKUP TABLE\n\n");

        /* Print column headers. */
        printf("    ");
        for (i = 0; i < 16; i++)
                printf("%X  ", i);
        fputchar('\n');

        /* Print each line of the table. */
        for (i = 0, k = 0; i < 16; i++)
                {
                /* Print first hex digit of symbols on this line. */
                printf("%X   ", i);
                /* Print each of the 16 symbols for this line. */
                for (j = 0; j < 16; j++)
                        {
                        /* Filter nonprintable characters. */
                        if ((k >= 7 && k <= 13) || (k >= 28 && k <= 31))
                                printf("   ");
                        else
                                printf("%c  ", k);
                        k++;
                        }
                fputchar('\n');
                }
}
```

Figure 18-17. The correct ASCII table generation program.

CodeView is a good choice for debugging C, Pascal, BASIC, and FORTRAN programs. The fact that versions of MASM earlier than 5.0 do not generate data for CodeView makes CodeView a poorer choice for these assembly-language programs. These disadvantages must be weighed against the ability to set watchpoints and to trap nonmaskable interrupts (NMIs). CodeView is also not as well suited as SYMDEB for debugging programs that interact with TSRs and device drivers, because CodeView does not provide any mechanism for including symbol tables for routines not linked together.

Hardware debugging aids

Hardware debuggers are a combination of hardware and software designed to be installed in a PC system. The software provides features much like those available with SYMDEB and CodeView. The advantages of hardware debuggers over purely software debuggers can be summarized in three points:

- Crash protection
- Manual execution break
- Hardware breakpoints

A hardware debugger can provide program crash protection because of its independence from the PC software. If the program being debugged goes wild and destroys the operating system of the PC, the hardware debugger is protected by virtue of being a separate hardware system and is capable of recovering enough control to allow the user to find out what happened.

All hardware debuggers offer a means of breaking into the program under test from some external source — usually a push button in the hands of the programmer. The mechanism used to get the attention of the PC's CPU is the nonmaskable interrupt (NMI). This interrupt provides a more reliable means of interrupting program execution than the Break key because its operation is independent of the state of interrupts and other conditions.

Hardware debuggers usually have access to the address and data lines on the PC bus, allowing them to set *hardware* breakpoints. Thus, these debuggers can be set to break when specific addresses are referenced. They execute the breakpoint code from a debugging monitor, which generally runs from their own memory. This memory is usually protected from the regular operating system and the application program.

Although hardware debuggers can be used to instrument a program, they should not be confused with the external hardware instrumentation discussed earlier in this article. The logic analyzers and in-circuit emulators mentioned there are general-purpose test instruments; the hardware debuggers are highly specific devices intended to do only one thing on one type of hardware — provide debugging monitor functions at a hardware level to IBM PC-type machines. It is this specialization that makes hardware debuggers so much easier to use for programmers trying to get a piece of code running.

Because this volume deals only with MS-DOS and associated Microsoft software, a detailed discussion of hardware debuggers and debugging would not be appropriate. Instead, a few popular hardware products that work with MS-DOS utilities are mentioned and a general discussion of debugging with hardware is presented.

Several manufacturers make hardware products that can be used for debugging. These products vary in the features offered and in their suitability for various kinds of debugging. Three of these products that can be used with SYMDEB are

- IBM Professional Debug Utility
- PC Probe and AT Probe from Atron Corporation
- Periscope from The Periscope Company, Inc.

These boards can be used with SYMDEB by specifying the /N switch when the program is started. When used in this way, however, the hardware provides little more than a source of NMIs to interrupt program execution; otherwise, SYMDEB runs as usual. This restriction may not be acceptable to a programmer who wants to use the sophisticated debugging software that accompanies these products and makes use of their hardware features. For this reason, these boards are rarely used with SYMDEB.

The general techniques of debugging with hardware aids will already be familiar to the reader—they are the same techniques discussed at length earlier in this article. The techniques of inspection and observation should still be applied; instrumentation is facilitated by hardware; a debugging monitor accompanies all hardware debuggers and the same techniques discussed for DEBUG, SYMDEB, and CodeView apply. No new techniques are needed to use these devices. The changes in the details of the techniques come with the added features available with the hardware debuggers. (Remember that all these features are not universally available on all hardware debuggers.)

The manual interrupt feature of hardware debuggers is useful in a system crash. Every programmer, especially assembly-language programmers, has had the situation where the program runs wild, destroys the operating system, and locks up the system. The techniques described in previous sections of this article show that about the only way to solve these problems without hardware help is to set breakpoints at strategic locations in the program and see how many are passed before the system locks up. The breakpoints are placed at finer and finer increments until the instruction causing the crash is located.

This long and ugly procedure can sometimes be shortened with a hardware debugger. When the system crashes, the programmer can push the manual interrupt button, suspend program execution, and give control to the debugger card. At this point, the programmer can use the debugging monitor software supplied with the card to sniff around memory looking for something suspicious. Clues can sometimes be found by examining the program's stack and data areas—provided, of course, that they are still in memory and haven't been destroyed, along with the operating system, by the rampaging program. This approach is not always an immediate solution to the problem, however; often, the start-and-set-breakpoints process has to be repeated even with a hardware debugger. The hardware will, however, possibly shed some light on the causes of the problem and shorten the procedure.

Another feature offered by many of the debugging boards is the ability to set breakpoints on events other than the execution of a line of code. Often, these boards will allow the programmer to break on a reference to a specific memory location, to a range of memory

locations, or to an I/O port. This feature allows a watch to be set on data, analogous to the watchpoint feature of CodeView. This technique is almost always useful, as it is with CodeView, but there is one class of problems where it is *essential* to reaching a solution.

Consider the case of a program that seems to be running well. Every so often, however, an ampersand appears in the middle of a payroll amount, or occasionally the program makes an erroneous branch and executes the wrong path. Suppose that, after painstaking investigation, the programmer discovers that these problems are being caused by a change in a specific location in memory sometime during the execution of the program. In debugging, the discovery of the cause of a problem usually leads almost instantly to a fix. Not so in this case. That byte of memory could be changed by an error in the program, by a glitch in the operating system or in a device driver, or by cosmic rays from outer space. Discovering the culprit in a case like this is almost impossible without the help of hardware breakpoints. Setting a breakpoint on the affected memory location and running the program will solve the problem. As soon as the memory location is changed, the breakpoint will be executed and the state of the system registers will point a clear finger at the instruction that caused the problem.

Hardware debuggers can provide significant aid to the serious programmer. They are especially helpful in debugging operating systems and operating-system services such as device drivers. They are also helpful in complicated situations where many programs may be running at the same time. The consensus among programmers who have hardware debuggers is that they are well worth the money.

Summary

Although Microsoft and others have provided an impressive array of technology to aid in program debugging, the most important tool a programmer has is his or her native wit and talent. As the examples in this article have illustrated, the technology makes the task easier, but never easy. In all cases, however, it is the programmer who debugs the program and solves the problems.

Technology will never be able to replace the person for solving the problem of a bug-ridden program. (This is an area where artificial intelligence will undoubtedly fail.) Therefore, it is the skills discussed in the first part of this article — debugging by inspection and observation — that deserve the greatest attention and practice. All the other techniques and technologies, with their ever-increasing sophistication, are only extensions of these basic techniques. A programmer who can debug effectively at the lowest level of technology will always be ready to use whatever advanced technology is available.

Therefore, as a final word, remember the rule that opened this article:

Gather enough information and the solution will be obvious.

All the rest of this article was merely a discussion of ways to gather the information.

Steve Bostwick

Article 19
Object Modules

Object modules are used primarily by programmers. The end user of an MS-DOS application need never be concerned with object code, object modules, and object libraries because application programs are almost always distributed as .EXE or .COM files that can be executed with a simple startup command.

An application programmer writing in a high-level language can use object modules and object libraries without knowing either the format of object code or the details of what the utilities that process object modules, such as the Microsoft Library Manager (LIB) and the Microsoft Object Linker (LINK), are actually doing. Most application programmers simply regard the contents of an object module as a "black box" and trust their compilers and object module utility programs to do the right thing.

A programmer using assembly language or an assembly-language debugger such as DEBUG or SYMDEB, however, might want to know more about the content and function of object modules. The use of assembly language gives the programmer more control over the actual contents of object modules, so knowing how the modules are constructed and examining their contents can sometimes help with program debugging.

Finally, a programmer writing a compiler, an assembler, or a language translator must know the details of object module format and processing. To take advantage of LIB and LINK, a language translator must construct object modules that conform to the format and usage conventions specified by Microsoft.

Note: This article assumes some background knowledge of the process by which source code is converted into an executable file in the MS-DOS environment. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: Structure of an Application Program; PROGRAMMING TOOLS: The Microsoft Object Linker; PROGRAMMING UTILITIES.

The Use of Object Modules

Although some MS-DOS language translators generate executable 8086-family machine code directly from source code, most produce object code instead. Typically, a translator processes each file of source code individually and leaves the resulting object module in a separate file bearing a .OBJ extension. The source-code files themselves remain unchanged. After all of a program's source-code modules have been translated, the resulting object modules can be linked into a single executable program. Because object modules frequently represent only a portion of a complete program, each source-code module usually contains instructions that indicate how its corresponding object code is to be combined with the object code in other object modules when they are linked.

The object code contained in each object module consists of a binary image of the program plus program structure information. This object code is not directly executable. The binary image corresponds to the executable code that will ultimately be loaded into memory for execution; it contains both machine code and program data. The program structure information includes descriptions of logical groupings defined in the source code (such as named subroutines or segments) and symbolic references to addresses in other object modules.

The program structure information is used by a linkage editor, or linker, such as Microsoft LINK to edit the binary image of the program contained in the object module. The linker combines the binary images from one or more object modules into a complete executable program.

The linker's output is a .EXE file—a file containing executable machine code that can be loaded into RAM and executed (Figure 19-1). The linker leaves intact all of the object modules it processes.

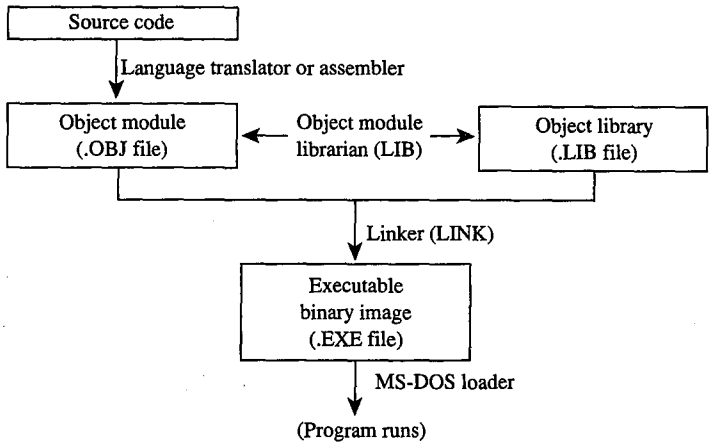


Figure 19-1. Generation of an executable (.EXE) file.

Object code thus serves as an intermediate form for compiled programs. This form offers two major advantages:

- Modular intermediate code. The use of object modules eliminates the overhead of repeated compilation of an entire program whenever changes are made to parts of its source code. Instead, only those object modules affected by source-code revisions need be recompiled.
- Shareable format. Object module format is well defined, so object modules can be linked even if they were produced by different translators. Many high-level-language compilers take advantage of this commonality of object-code format to support "interlanguage" linkage.

Contents of an object module

Object modules contain five basic types of information. Some of this information exists explicitly in the source code (and is subsequently passed on to the object module), but much is inferred by the program translator from the structure of the source code and the way memory is accessed by the 8086.

Binary Image. As described earlier, the binary image comprises executable code (such as opcodes and addresses) and program data. When object modules are linked, the linker builds an executable program from the binary image in each object module it processes. The binary image in each object module is always associated with program structure information that tells the linker how to combine it with related binary images in other object modules.

External References. Because an object module generally represents only a small portion of a larger program that will be constructed from several object modules, it usually contains symbols that allow it to be linked to the other modules. Such references to corresponding symbols in other object modules are resolved when the modules are linked.

For example, consider the following short C program:

```
main()
{
        puts("Hello, world\n");
}
```

This program calls the C function *puts()* to display a character string, but *puts()* is not defined in the source code. Rather, the name *puts* is a reference to a function that is external to the program's *main()* routine. When the C compiler generates an object module for this program, it will identify *puts* as an external reference. Later, the linker will resolve the external reference by linking the object module containing the *puts()* routine with the module containing the *main()* routine.

Address References. When a program is built from a group of object modules, the actual values of many addresses cannot be computed until the linker combines the binary image of executable code and the program data from each of the program's constituent object modules. Object modules contain information that tells the linker how to resolve the values of such addresses, either symbolically (as in the case of external references) or relatively, in terms of some other address (such as the beginning of a block of executable code or program data).

Debugging Information. An object module can also contain information that relates addresses in the executable program to the corresponding source code. After the linker performs its address fixups, it can use the object module's debugging information to relate a line of source code in a program module to the executable code that corresponds to it.

Miscellaneous Information. Finally, an object module can contain comments, lists of symbols defined in or referenced by the module, module identification information, and

information for use by an object library manager or a linker (for example, the names of object libraries to be searched by default).

Object module terminology

When the linker generates an executable program, it organizes the structural components of the program according to the information contained in the object modules. The layout of the executable program can be conceptually described as a run-time memory map after it has been loaded into memory.

The basic structure of every executable program for the 8086 family of microprocessors must conform to the segmented architecture of the microprocessor. Thus, the run-time memory map of an executable program is partitioned into segments, each of which can be addressed by using one of the microprocessor's segment registers. This segmented structure of 8086-based programs is the basis for most of the following terminology.

Frames. The memory address space of the 8086 is conceptually divided into a sequence of paragraph-aligned, overlapping 64 KB regions called frames. Frame 0 in the 8086's address space is the 64 KB of memory starting at physical address 00000H (0000:0000 in segment:offset notation), frame 1 is the 64 KB of memory starting at 00010H (0001:0000), and so on. A frame number thus denotes the beginning of any paragraph-aligned 64 KB of memory. For example, the location of a 64 KB buffer that starts at address B800:0000 can be specified as frame 0B800H.

Logical Segments. The run-time memory map for every 8086 program is partitioned into one or more logical segments, which are groupings of logically related portions of the program. Typically, an MS-DOS program includes at least one code segment (that contains all of the program's executable code), one or more data segments (that contain program data), and one stack segment.

When a program is loaded into RAM to be executed, each logical segment in the program can be addressed with a frame number—that is, a physical 8086 segment address. Before the MS-DOS loader transfers control to a program in memory, it initializes the CS and SS registers with the segment addresses of the program's executable code and stack segments. If an MS-DOS program has a separate logical segment for program data, the program itself usually stores this segment's address in the DS register.

Relocatable Segments. In MS-DOS programs, most logical segments are relocatable. The loader determines the physical addresses of a program's relocatable segments when it places the program into memory to be executed. However, this address determination poses a problem for the MS-DOS loader, because a program may contain references to the address of a relocatable segment even though the address value is not determined until the program is loaded. The problem is solved by indicating where such references occur within the program's object modules. The linker then extracts this information from the object modules and uses it to build a list of such address references into a segment relocation table in the header of executable files. After the loader copies a program into memory for execution, it uses the segment relocation table to update, or fix up, the segment address references within the program.

Consider the following example, in which a program loads the starting addresses of two data segments into the DS and ES segment registers:

```
mov     ax,seg _DATA
mov     ds,ax               ; make _DATA segment addressable through DS
mov     ax,seg FAR_DATA
mov     es,ax               ; make FAR_DATA segment addressable through ES
```

The actual addresses of the _DATA and FAR_DATA segments are unknown when the source code is assembled and the corresponding object module is constructed. The assembler indicates this by including segment fixup information, instead of actual segment addresses, in the program's object module. When the object module is linked, the linker builds this segment fixup information into the segment relocation table in the header of the program's .EXE file. Then, when the .EXE file is loaded, the MS-DOS loader uses the information in the .EXE file's header to patch the actual address values into the program.

Absolute Segments. Sometimes a program needs to address a predetermined segment of memory. In this case, the program's source code must declare an absolute segment so that a reference to the corresponding frame number can be built into the program's object module.

For example, a program might need to address a video display buffer located at a specific physical address. The following assembler directive declares the name of the segment and its frame number:

```
VideoBufferSeg    SEGMENT at 0B800h
```

Segment Alignment. When a program is loaded, the physical address of each logical segment is constrained by the segment's alignment. A segment can be page aligned (aligned on a 256-byte boundary), paragraph aligned (aligned on a 16-byte paragraph boundary), word aligned (aligned on an even-byte boundary), or byte aligned (not aligned on any particular boundary). A specification of each segment's alignment is part of every object module's program structure information.

High-level-language translators generally align segments according to the type of data they contain. For example, executable code segments are usually byte aligned; program data segments are usually word aligned. With an assembler, segment alignment can be specified with the SEGMENT directive and the assembler will build this information into the program's object module.

Concatenated Segments. The linker can concatenate logical segments from different object modules when it builds the executable program. For example, several object modules may each contain part of a program's executable code. When the linker processes these object modules, it can concatenate the executable code from the different object modules into one range of contiguous addresses.

The order in which the linker concatenates logical segments in the executable program is determined by the order in which the linker processes its input files and by the program

structure information in the object modules. With a high-level-language translator, the translator infers which segments can be concatenated from the structure of the source code and builds appropriate segment concatenation information into the object modules it generates. With an assembler, the segment class type can be used to indicate which segments can be concatenated.

Groups of Segments. Segments with different names may also be grouped together by the linker so that they can all be addressed within the same 64 KB frame, even though they are not concatenated. For example, it might be desirable to group program data segments and a stack segment within the same 64 KB frame so that program data items and data on the stack can be addressed with the same 8086 segment register.

In high-level languages, it is up to the translator to incorporate appropriate segment grouping information into the object modules it generates. With an assembler, groups of segments can be declared with the GROUP directive.

Fixups. Sometimes a compiler or an assembler encounters addresses whose values cannot be determined from the source code. The addresses of external symbols are an obvious example. The addresses of relocatable segments and of labels within those segments are another example.

A fixup is a language translator's way of passing the buck about such addresses to the linker. Typically, a translator builds a zero value in the binary image at locations where it cannot store an actual address. Accompanying each such location is fixup information, which allows the linker to determine the correct address. The linker then completes the fixup by calculating the correct address value and adding it to the value in the corresponding location in the binary image. The only fixups the linker cannot fully resolve are those that refer to the segment address of a relocatable segment. Such addresses are not known until the program is actually loaded, so the linker, in turn, passes the responsibility to the MS-DOS loader by creating a segment relocation table in the header of the executable file.

To process fixups properly, the linker needs three pieces of information: the LOCATION of the value in the object module, the nature of the TARGET (the address whose value is not yet known), and the FRAME in which the address calculations are to take place. Object modules contain the LOCATION, TARGET, and FRAME information the linker uses to calculate the appropriate address for any given fixup.

Consider the "program" in Figure 19-2. The statement:

```
start:  call    far ptr FarProc
```

contains a reference to an address in the logical segment *FarSeg2*. Because the assembler does not know the address of *FarSeg2*, it places fixup information about the address into the object module. The LOCATION to be fixed up is 1 byte past the label *start* (the 4-byte pointer following the *call* opcode 9AH). The TARGET is the address referenced in the *call* instruction—that is, the label *FarProc* in the segment *FarSeg2*. The FRAME to which

the fixup relates is designated by the group *FarGroup* and is inferred from the statement

```
ASSUME  cs:FarGroup
```

in the *FarSeg2* segment.

```
                                        title   fixups


                                FarGroup GROUP  FarSeg1,FarSeg2

0000                            CodeSeg SEGMENT byte public 'CODE'
                                        ASSUME  cs:CodeSeg

0000  9A 0000 ---- R            start:  call    far ptr FarProc

0005                            CodeSeg ENDS


0000                            FarSeg1 SEGMENT byte public     ;part of FarGroup

0000                            FarSeg1 ENDS


0000                            FarSeg2 SEGMENT byte public
                                        ASSUME  cs:FarGroup

0000                            FarProc PROC    far
0000  CB                                ret                     ;a FAR return
                                FarProc ENDP

0001                            FarSeg2 ENDS

                                        END
```

Figure 19-2. A sample "program" containing statements from which the assembler derives fixup information.

There are several different ways for a language translator to identify a fixup. For example, the LOCATION might be a single byte, a 16-bit offset, or a 32-bit pointer, as in Figure 19-2. The TARGET might be a label whose offset is relative either to the base (beginning) of a particular segment or to the LOCATION itself. The FRAME might be a relocatable segment, an absolute segment, or a group of segments.

Taken together, all the information in an object module that concerns the alignment and grouping of segments can be regarded as a specification of a program's run-time memory map. In effect, the object module specifies what goes where in memory when a program is loaded. The linker can then take the program structure information in the object modules and generate a file containing an executable program with the corresponding structure.

The Structure of an Object Module

Although object modules contain the information that ultimately determines the structure of an executable program, they bear little structural resemblance to the resulting executable program. Each object module is made up of a sequence of variable-length object records. Different types of object records contain different types of program information.

Each object record begins with a 1-byte field that identifies its type. This is followed by a 2-byte field containing the length (in bytes) of the remainder of the record. Next comes the actual structural or program information, represented in one or more fields of varied lengths. Finally, each record ends with a 1-byte checksum.

The sequence in which object records appear in an object module is important. Because the records vary in length, each object module must be constructed linearly, from start to end. More important, however, is the fact that some types of object records contain references to preceding object records. Because the linker processes object records sequentially, the position of each object record within an object module depends primarily on the type of information each record contains.

Types of object records

Microsoft LINK currently recognizes 14 types of object records, each of which carries a specific type of information within the object module. Each type of object record is assigned an identifying six-letter abbreviation, but these abbreviations are used only in documentation, not within an object module itself. As already mentioned, the first byte of each object record contains a value that indicates its type. In a hexadecimal dump of the contents of an object module, these identifying bytes identify the start of each object record.

Table 19-1 lists the types of object records supported by LINK. The value of each record's identifying byte (in hexadecimal) is included, along with the six-letter abbreviation and a brief functional description. The functions of the 14 types of object records fall into six general categories:

- Binary data (executable code and program data) is contained in the LEDATA and LIDATA records.
- Address binding and relocation information is contained in FIXUPP records.
- The structure of the run-time memory map is indicated by SEGDEF, GRPDEF, COMDEF, and TYPDEF records.
- Symbol names are declared in LNAMES, EXTDEF, and PUBDEF records.
- Debugging information is in the LINNUM record.
- Finally, the structure of the object module itself is determined by the THEADR, COMENT, and MODEND records.

Table 19-1. Types of 8086 Object Records Supported by Microsoft LINK.

ID byte	Abbreviation	Description
80H	THEADR	Translator Header Record
88H	COMENT	Comment Record
8AH	MODEND	Module End Record
8CH	EXTDEF	External Names Definition Record
8EH	TYPDEF	Type Definition Record
90H	PUBDEF	Public Names Definition Record
94H	LINNUM	Line Number Record
96H	LNAMES	List of Names Record
98H	SEGDEF	Segment Definition Record
9AH	GRPDEF	Group Definition Record
9CH	FIXUPP	Fixup Record
0A0H	LEDATA	Logical Enumerated Data Record
0A2H	LIDATA	Logical Iterated Data Record
0B0H	COMDEF	Communal Names Definition Record

Object record order

The sequence in which the types of object records appear in an object module is fairly flexible in some respects. Several record types are optional, and if the type of information they carry is unnecessary, they are omitted from an object module. In addition, most object record types can occur more than once in the same object module. And, because object records are variable in length, it is often possible to choose, as a matter of convenience, between combining information into one large record or breaking it down into several smaller records of the same type.

As stated previously, an important constraint on the order in which object records appear is the need for some types of object records to refer to information contained in other records. Because the linker processes the records sequentially, object records containing such information must precede the records that refer to it. For example, two types of object records, SEGDEF and GRPDEF, refer to the names contained in an LNAMES record. Thus, an LNAMES record must appear before any SEGDEF or GRPDEF records that refer to it so that the names in the LNAMES record are known to the linker by the time it processes the SEGDEF or GRPDEF records.

A typical object module

Figure 19-3 contains the source code for HELLO.ASM, an assembly-language program that displays a short message. Figure 19-4 is a hexadecimal dump of HELLO.OBJ, the object module generated by assembling HELLO.ASM with the Microsoft Macro Assembler. Figure 19-5 isolates the object records within the object module.

```
        NAME    HELLO

_TEXT   SEGMENT byte public 'CODE'

        ASSUME  cs:_TEXT,ds:_DATA

start:                                  ;program entry point
        mov     ax,seg msg
        mov     ds,ax
        mov     dx,offset msg           ;DS:DX -> msg
        mov     ah,09h
        int     21h                     ;perform int 21H function 09H
                                        ;(Output character string)
        mov     ax,4C00h
        int     21h                     ;perform int 21H function 4CH
                                        ;(Terminate with return code)
_TEXT   ENDS


_DATA   SEGMENT word public 'DATA'

msg     DB      'Hello, world',0Dh,0Ah,'$'

_DATA   ENDS


_STACK  SEGMENT stack 'STACK'

        DW      80h dup(?)              ;stack depth = 128 words

_STACK  ENDS

        END     start
```

Figure 19-3. The source code for HELLO.ASM.

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  80 07 00 05 48 45 4C 4C 4F 00 96 25 00 00 04 43  ....HELLO..%...C
0010  4F 44 45 04 44 41 54 41 05 53 54 41 43 4B 05 5F  ODE.DATA.STACK._
0020  44 41 54 41 06 5F 53 54 41 43 4B 05 5F 54 45 58  DATA._STACK._TEX
0030  54 8B 98 07 00 28 11 00 07 02 01 1E 98 07 00 48  T....(.........H
0040  0F 00 05 03 01 01 98 07 00 74 00 01 06 04 01 E1  .........t......
0050  A0 15 00 01 00 00 B8 00 00 8E D8 BA 00 00 B4 09  ................
0060  CD 21 B8 00 4C CD 21 D5 9C 0B 00 C8 01 04 02 02  .!..L.!.........
0070  C4 06 04 02 02 B6 A0 13 00 02 00 00 48 65 6C 6C  ............Hell
0080  6F 2C 20 77 6F 72 6C 64 0D 0A 24 A8 8A 07 00 C1  o, world..$.....
0090  00 01 01 00 00 AC                                ......
```

Figure 19-4. A hexadecimal dump of HELLO.OBJ.

```
       0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F

THEADR
0000  80 07 00 05 48 45 4C 4C 4F 00                    ....HELLO.

LNAMES
0000                                 96 25 00 00 04 43           .%...C
0010  4F 44 45 04 44 41 54 41 05 53 54 41 43 4B 05 5F  ODE.DATA.STACK._
0020  44 41 54 41 06 5F 53 54 41 43 4B 05 5F 54 45 58  DATA._STACK._TEX
0030  54 8B                                           T.

SEGDEF
0030        98 07 00 28 11 00 07 02 01 1E                ...(......

SEGDEF
0030                                       98 07 00 48              ...H
0040  0F 00 05 03 01 01                                ......

SEGDEF
0040                    98 07 00 74 00 01 06 04 01 E1        ...t......

LEDATA
0050  A0 15 00 01 00 00 B8 00 00 8E D8 BA 00 00 B4 09  ................
0060  CD 21 B8 00 4C CD 21 D5                          .!..L.!.

FIXUPP
0060                           9C 0B 00 C8 01 04 02 02          ........
0070  C4 06 04 02 02 B6                                ......

LEDATA
0070                     A0 13 00 02 00 00 48 65 6C 6C        ......Hell
0080  6F 2C 20 77 6F 72 6C 64 0D 0A 24 A8              o, world..$.

MODEND
0080                                       8A 07 00 C1              ....
0090  00 01 01 00 00 AC                                ......
```

Figure 19-5. The object records in HELLO.OBJ.

As shown most clearly in Figure 19-5, each of the object records begins with the single byte value identifying the record's type. The second and third bytes of each record contain a single 16-bit value, stored with its low-order byte first, that represents the length (in bytes) of the remainder of the object record.

The first record, THEADR, identifies the object module and the last record, MODEND, terminates the object module. The second record, LNAMES, contains a list of segment names and segment class names that LINK will use to lay out the run-time memory map. The three succeeding SEGDEF records describe the three corresponding segments defined in the source code.

The order in which the object records appear reflects both the structure of the source code and the record order constraints already mentioned. The LNAMES record appears before the three SEGDEF records because each SEGDEF record contains a reference to a name in the LNAMES record.

The binary data representing each of the two segments in the source code is contained in the two LEDATA records. The first LEDATA record represents the *_TEXT* segment; the second specifies the data in the *_DATA* segment. The FIXUPP record following the first LEDATA record contains information about the address references in the *_TEXT* segment. Again, the order in which the records appear is important: the FIXUPP record refers to the LEDATA record preceding it.

References between object records

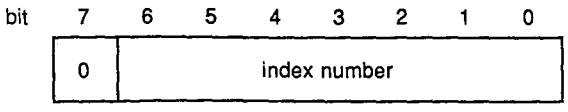
Object records can refer to information in other records either indirectly, by means of implicit references, or directly, by means of indexed references to names or other records.

Implicit References. Some types of object records implicitly reference another record in the same object module. The most important example of such implicit referencing is in the FIXUPP record, which always contains fixup information for the preceding LEDATA or LIDATA record in the object module. Whenever an LEDATA or LIDATA record contains a value that needs to be fixed up, the next record in the object module is always a FIXUPP record containing the actual fixup information.

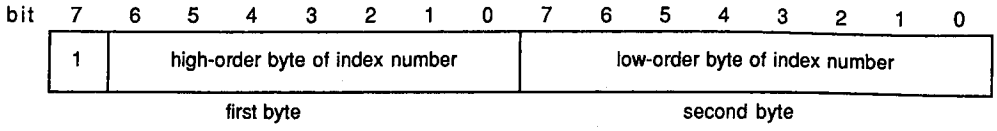
Indexed References to Names. An object record that refers to a symbolic name, such as the name of a segment or an external routine, uses an index into a list of names contained in a previous object record. (The LNAMES record in Figure 19-5 is an example.) The first name in such a list has the index number 1, the second name has index number 2, the third has index number 3, and so on. Altogether, a list of as many as 32,767 (7FFFH) names can be incorporated into an object module—generally adequate for even the most verbose programmer. (LINK does, however, impose its own version-specific limits.)

Indexed References to Object Records. An object record can also refer to a previous object record by using the same type of index. In this case, the index number refers to one of a list of object records of a particular type. For example, a FIXUPP record might refer to a segment by referencing one of several preceding SEGDEF records in the object module. In that case, a value of 1 would indicate the first SEGDEF record in the object module, a value of 2 would indicate the second, and so on.

The *index-number* field in an object record can be either 1 or 2 bytes long. If the number is in the range 0–7FH, the high-order bit (bit 7) is 0 and the low-order 7 bits contain the index number, so the field is only 1 byte long:



If the index number is in the range 80–7FFFH, the field is 2 bytes long. The high-order bit of the first byte in the field is set to 1, and the high-order byte of the index number (which must be in the range 0–7FH) fits in the remaining 7 bits. The low-order byte of the index number is specified in the second byte of the field:



The same format is used whether an index refers to a list of names or to a previous object record.

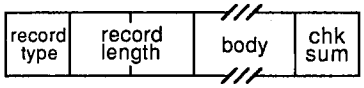
Microsoft 8086 Object Record Formats

Just as the design of the Intel 8086 microprocessor reflects the design of its 8-bit predecessors, 8086 object record formats are reminiscent of the 8-bit software tradition. In 8-bit systems, disk space and RAM were often at a premium. To minimize the space consumed by object records, information is packed into bit fields within bytes and variable-length fields are frequently used.

Microsoft LINK recognizes a major subset of Intel's original 8086 object module specification (Intel Technical Specification 121748-001). Intel also proposed a six-letter name for each type of object record and symbolic names for fields. These names are documented in the following descriptions, which appear in the order shown earlier in Table 19-1.

The Intel record types that are not recognized by LINK provide information about an executable program that MS-DOS obtains in other ways. (For example, information about run-time overlays is supplied in LINK's command line rather than being encoded in object records.) Because they are ignored by LINK, they are not included here.

All 8086 object records conform to the following format:



The *record type* field is a 1-byte field containing the hexadecimal number that identifies the type of object record (*see* Table 19-1).

The *record length* is a 2-byte field that gives the length of the remainder of the object record in bytes (excluding the bytes in the *record type* and *record length* fields). The record length is stored with the low-order byte first.

The *body* field of the record varies in size and content, depending on the record type.

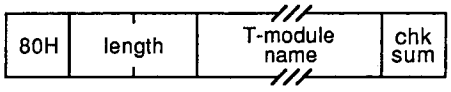
The *checksum* is a 1-byte field that contains the negative sum (modulo 256) of all other bytes in the record. In other words, the checksum byte is calculated so that the low-order byte of the sum of all the bytes in the record, including the checksum byte, equals zero.

Note: As shown in the preceding example, the boxes used to depict the fields vary in size. The square boxes used for *record type* and *chksum* indicate a single byte, the rectangular box used for *record length* indicates 2 bytes, and the diagonal lines used for *body* indicate a variable-length field.

80H THEADR Translator Header Record

The THEADR record contains the name of the object module. This name identifies an object module within an object library or in messages produced by the linker.

Record format



T-module name

The *T-module name* field is a variable-length field that contains the name of the object module. The first byte of the field contains the number of subsequent bytes that contain the name itself. The name can be uppercase or lowercase and can be any string of characters.

The *T-module name* is used by LIB and LINK within error messages. Language translators frequently derive the *T-module name* from the name of the file that contains a program's source code. Assembly-language programmers can specify the *T-module name* explicitly with the assembler NAME directive.

Location in object module

As its name implies, the THEADR record must be the first record in every object module generated by a language translator.

Example

The following THEADR record was generated by the Microsoft C Compiler:

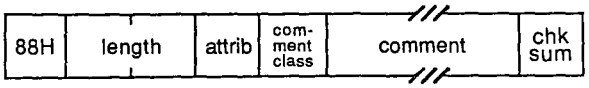
```
       0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  80 09 00 07 68 65 6C 6C 6F 2E 63 CB              ....hello.c.
```

- Byte 00H contains 80H, indicating a THEADR record.
- Bytes 01–02H contain 0009H, the length of the remainder of the record.
- Bytes 03–0AH contain the *T-module name*. Byte 03H contains 07H, the length of the name, and bytes 04H through 0AH contain the name itself (*hello.c*). (In object modules generated by the Microsoft C Compiler, the THEADR record indicates the filename that contained the C source code for the module.)
- Byte 0BH contains the checksum, 0CBH.

88H COMENT Comment Record

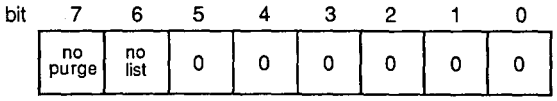
The COMENT record contains a character string that may represent a plain text comment, a symbol meaningful to a program such as LIB or LINK, or even binary-encoded identification data. An object module can contain any number of COMENT records.

Record format



Attrib

Attrib is a 1-byte field in which only the first 2 bits are meaningful:



- If bit 7 (*no purge*) is set to 1, utility programs that manipulate object modules should not delete the comment record from the object module. Bit 7 can thus protect an important comment, such as a copyright message, from deletion.
- If bit 6 (*no list*) is set to 1, utility programs that can list the contents of object modules are directed not to list the comment. Bit 6 can thus hide a comment.
- Bits 5 through 0 are unused and should be set to 0.

Microsoft LIB ignores the *attrib* field.

Comment class

Comment class is a 1-byte field whose value provides information about the type of comment. The original Intel specification provided for the following possible *comment class* values:

Value	Use
00H	Language-translator comment (the name of the translator that generated the object module).
01H	Copyright comment.
02–9BH	Reserved for Intel proprietary software.

Microsoft language translators can generate several other classes of COMENT record that communicate specific information about the object module to LINK:

Value	Use
81H	Obsolete; replaced by *comment class* 9FH.
9CH	MS-DOS version number. Some language translators create a COMENT record with a 2-byte binary value in the *comment* field indicating the MS-DOS version under which the module was created. This record is ignored by LINK.
9DH	Memory model. The *comment* field contains a string that indicates the memory model used by the language translator. The string contains one of the lowercase letters s, c, m, l, and h to designate small, compact, medium, large, and huge memory models. Microsoft language translators generate COMENT records with this *comment class* only for compatibility with the XENIX version of LINK. The MS-DOS version of LINK ignores these COMENT records.
9EH	Sets Microsoft LINK's DOSSEG switch.
9FH	Default library search name. LINK interprets the contents of the *comment* field as the name of a library to be searched in order to resolve external references within the object module. The default library search can be overridden with LINK's NODEFAULTLIBRARYSEARCH switch.
0A1H	Indicates that Microsoft extensions to the Intel object record specification are used in the object module. For example, when COMDEF records are used within an object module, a COMENT record with *comment class* 0A1H must appear in the object module at some point before the first COMDEF record. LINK ignores the *comment* string in COMENT records with this *comment class.*
0C0H–0FFH	Reserved for user-defined comment classes.

Comment

The *comment* field is a variable-length string of bytes that represent the comment. The length of the string is inferred from the length of the object record.

Location in object module

A COMENT record can appear almost anywhere in an object module. Only two restrictions apply:

- A COMENT record cannot be placed between a FIXUPP record and the LEDATA or LIDATA record to which it refers.
- A COMENT record cannot be the first or last record in an object module. (The first record must always be a THEADR record and the last must always be MODEND.)

Examples

The following three examples are typical COMENT records taken from an object module generated by the Microsoft C Compiler.

This first example is a language-translator comment:

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  88 07 00 00 00 4D 53 20 43 6E                  .....MS Cn
```

- Byte 00H contains 88H, indicating that this is a COMENT record.
- Bytes 01–02H contain 0007H, the length of the remainder of the record.
- Byte 03H (the *attrib* field) contains 00H. Bit 7 (*no purge*) is set to 0, indicating that this COMENT record may be purged from the object module by a utility program that manipulates object modules. Bit 6 (*no list*) is set to 0, indicating that this comment need not be excluded from any listing of the module's contents. The remaining bits are all 0.
- Byte 04H (the *comment class* field) contains 00H, indicating that this COMENT record contains the name of the language translator that generated the object module.
- Bytes 05H through 08H contain the name of the language translator, MS C.
- Byte 09H contains the checksum, 6EH.

The second example contains the name of an object library to be searched by default when LINK processes the object module containing this COMENT record:

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  88 09 00 00 9F 53 4C 49 42 46 50 10            .....SLIBFP.
```

- Byte 04H (the *comment class* field) contains 9FH, indicating that this record contains the name of a library for LINK to use to resolve external references.
- Bytes 05–0AH contain the library name, SLIBFP. In this example, the name refers to the Microsoft C Compiler's floating-point function library, SLIBFP.LIB.

The last example indicates that the object module contains Microsoft-defined extensions to the Intel object module specification:

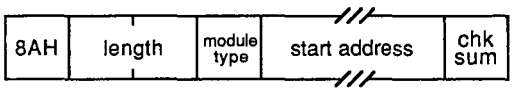
```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  88 06 00 00 A1 01 43 56 37                     .....CV7
```

- Byte 04H indicates the *comment class*, 0A1H.
- Bytes 05–07H, which contain the *comment* string, are ignored by LINK.

8AH MODEND Module End Record

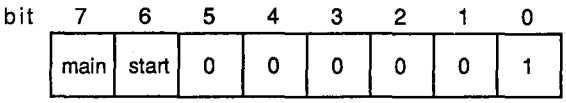
The MODEND record denotes the end of an object module. It also indicates whether the object module contains the main routine in a program, and it can, optionally, contain a reference to a program's entry point.

Record format



Module type

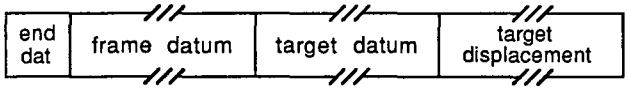
The *module type* field is an 8-bit (1-byte) field:



- Bit 7 (*main*) is set to 1 if the module is a main program module.
- Bit 6 (*start*) is set to 1 if the MODEND record contains an entry point (*start address*).
- Bit 0 is set to 1 if the *start address* field contains a relocatable address reference that LINK must fix up. If bit 6 is set to 1, bit 0 must also be set to 1. (The Intel specification allows bit 0 to be set to 0, to indicate that *start address* is an absolute physical address, but this capability is not supported by LINK.)

Start address

The *start address* field appears in the MODEND record only when bit 6 is set to 1:



The format and interpretation of the *start address* field corresponds to the *fixup* field of the FIXUPP record. The *end dat* field corresponds to the *fix dat* field in the FIXUPP record. Bit 2 of the *end dat* field, which corresponds to the *P* bit in a *fix dat* field, must be zero.

Location in object module

A MODEND record can appear only as the last record in an object module.

Example

Consider the *MODEND* record of the HELLO.ASM example:

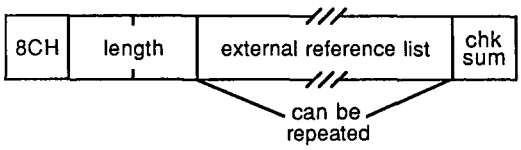
```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  8A 07 00 C1 00 01 01 00 00 AC                  ..........
```

- Byte 00H contains 8AH, indicating a MODEND record.
- Bytes 01–02H contain 0007H, the length of the remainder of the record.
- Byte 03H contains 0C1H (11000001B). Bit 7 is set to 1, indicating that this module is the main module of the program. Bit 6 is set to 1, indicating that a *start address* field is present. Bit 0 is set to 1, indicating that the address referenced in the *start address* field must be fixed up by LINK.
- Byte 04H (*end dat* in the *start address* field) contains 00H. As in a FIXUPP record, bit 7 indicates that the frame for this fixup is specified explicitly, and bits 6 through 4 indicate that a SEGDEF index specifies the frame. Bit 3 indicates that the target reference is also specified explicitly, and bits 2 through 0 indicate that a SEGDEF index also specifies the target. *See also* FIXUPP 9CH Fixup Record below.
- Byte 05H (*frame datum* in the *start address* field) contains 01H. This is a reference to the first SEGDEF record in the module, which in this example corresponds to the *_TEXT* segment. This reference tells LINK that the start address lies in the *_TEXT* segment of the module.
- Byte 06H (*target datum* in the *start address* field) contains 01H. This too is a reference to the first SEGDEF record in the object module, which corresponds to the *_TEXT* segment. LINK uses the following *target displacement* field to determine where in the *_TEXT* segment the address lies.
- Bytes 07–08H (*target displacement* in the *start address* field) contain 0000H. This is the offset (in bytes) of the *start address*.
- Byte 09H contains the checksum, 0ACH.

8CH EXTDEF External Names Definition Record

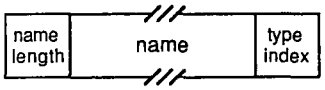
The EXTDEF record contains a list of symbolic external references — that is, references to symbols defined in other object modules. The linker resolves external references by matching the symbols declared in EXTDEF records with symbols declared in PUBDEF records.

Record format



External reference list

The *external reference list* is a variable-length field containing a list of names and name types, each formatted as follows:



- The *name length* is a 1-byte field containing the length of the *name* field that follows it. (LINK restricts *name length* to a value between 01H and 7FH.)
- The *type index* is a 1-byte reference to the TYPDEF record in the object module that describes the type of symbol the name represents. A *type index* value of zero indicates that no TYPDEF record is associated with the symbol. A nonzero value indicates which TYPDEF record is associated with the external name. Microsoft LINK recognizes TYPDEF records only for the purpose of declaring communal variables. *See* 8EH TYPDEF Type Definition Record below.

LINK imposes a limit of 1023 external names.

Location in object module

Any EXTDEF records in an object module must appear before the FIXUPP records that reference them. Also, if an EXTDEF record contains a nonzero *type index*, the indexed TYPDEF record must precede the EXTDEF record.

Example

Consider this EXTDEF record generated by the Microsoft C Compiler:

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  8C 25 00 0A 5F 5F 61 63 72 74 75 73 65 64 00 05  .%.._acrtused..
0010  5F 6D 61 69 6E 00 05 5F 70 75 74 73 00 08 5F 5F  _main.._puts.__
0020  63 68 6B 73 74 6B 00 A5                          chkstk..
```

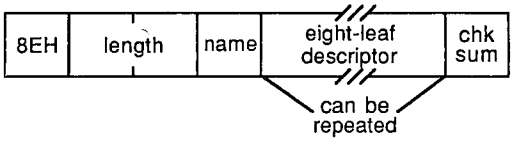
- Byte 00H contains 8CH, indicating that this is an EXTDEF record.
- Bytes 01–02H contain 0025H, the length of the remainder of the record.
- Bytes 03–26H contain a list of external references. The first reference starts in byte 03H, which contains 0AH, the length of the name *__acrtused*. The name itself follows in bytes 04–0DH. Byte 0EH contains 00H, which indicates that the symbol's type is not defined by any TYPDEF record in this object module. Bytes 0F–26H contain similar references to the external symbols *_main*, *_puts*, and *__chkstk*.
- Byte 27H contains the checksum, 0A5H.

8EH TYPDEF Type Definition Record

The TYPDEF record contains details about the type of data represented by a name declared in a PUBDEF or an EXTDEF record. This information may be used by the linker to validate references to names, or it may be used by a debugger to display data according to type.

Starting with Microsoft LINK version 3.50, the COMDEF record should be used for declaration of communal variables. For compatibility, however, later versions of LINK recognize TYPDEF records as well as COMDEF records.

Record format



Although the original Intel specification allowed for many different type specifications, such as scalar, pointer, and mixed data structure, LINK uses TYPDEF records to declare only communal variables. Communal variables represent globally shared memory areas—for example, FORTRAN common blocks or uninitialized public variables in C.

The size of a communal variable is declared explicitly in the TYPDEF record. If a communal variable has different sizes in different object modules, LINK uses the largest declared size when it generates an executable module.

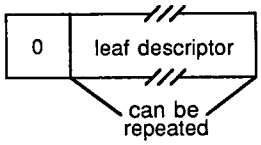
Name

The *name* field of a TYPDEF record is a 1-byte field that is always null; that is, it contains a single zero byte.

Eight-leaf descriptor

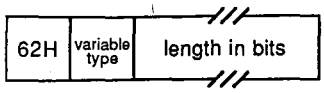
The *eight-leaf descriptor* field, in the original Intel specification, was a variable-length field that contained as many as eight "leaves" that could be used to describe mixed data structures.

Microsoft uses a stripped-down version of the *eight-leaf descriptor*, because the field's only function is to describe communal variables:



- The first field in the *eight-leaf descriptor* is a 1-byte field that contains a zero byte.
- The *leaf descriptor* field is a variable-length field that is itself divided into four fields ("leaves") that describe the size and type of a variable. The two possible variable types are NEAR and FAR.

  If the field describes a NEAR variable (one that can be referenced as an offset within a default data segment), the format is

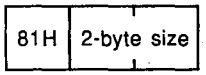


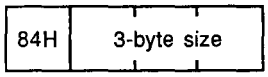
  - The 1-byte field containing 62H signifies a NEAR variable.
  - The *variable type* field is a 1-byte field that specifies the variable type:

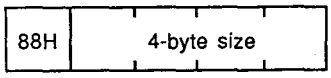
77H	Array
79H	Structure
7BH	Scalar

    This field is ignored by LINK.
  - The *length in bits* field is a variable-length field that indicates the size of the communal variable. Its format depends on the size it represents. If the size is less than 128 (80H) bits, *length in bits* is a 1-byte field containing the actual size of the field:

    If the size is 128 bits or greater, it cannot be represented in a single byte value, so the *length in bits* field is formatted with an extra initial byte that indicates whether the size is represented as a 2-, 3-, or 4-byte value:







If the *leaf descriptor* field describes a FAR variable (one that must be referenced with an explicit segment and offset), the format is

- The 1-byte field containing 61H signifies a FAR variable.
- The 1-byte *variable type* for a FAR communal variable is restricted to 77H (array). (As with the NEAR *variable type* field, LINK ignores this field.)
- The *number of elements* is a variable-length field that contains the number of elements in the array. It has the same format as the *length in bits* field in the *leaf descriptor* for a NEAR variable.
- The *element type index* is an index field that references a previous TYPDEF record. A value of 1 indicates the first TYPDEF record in the object module, a value of 2 indicates the second TYPDEF record, and so on. The TYPDEF record referenced must describe a NEAR variable. This way, the data type and size of the elements in the array can be determined.

Location in object module

Any TYPDEF records in an object module must precede the EXTDEF or PUBDEF records that reference them.

Examples

The following three examples of TYPDEF records were generated by the Microsoft C Compiler version 3.0. (Later versions use COMDEF records.)

The first sample TYPDEF record corresponds to the public declaration

```
int    foo;            /* 16-bit integer */
```

The TYPEDEF record is

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  8E 06 00 00 00 62 7B 10 7F                     .....b{..
```

- Byte 00H contains 8EH, indicating that this is a TYPDEF record.
- Bytes 01–02H contain 0006H, the length of the remainder of the record.
- Byte 03H (the *name* field) contains 00H, a null name.
- Bytes 04–07H represent the *eight-leaf descriptor* field. The first byte of this field (byte 04H) contains 00H. The remaining bytes (bytes 05–07H) represent the *leaf descriptor* field:
  - Byte 05H contains 62H, indicating this TYPDEF record describes a NEAR variable.
  - Byte 06H (the *variable type* field) contains 7BH, which describes this variable as a scalar.
  - Byte 07H (the *length in bits* field) contains 10H, the size of the variable in bits.

- Byte 08H contains the checksum, 7FH.

The next example demonstrates how the variable size contained in the *length in bits* field of the *leaf descriptor* is formatted:

```
char    foo2[32768];            /* 32 KB array */

       0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  8E 09 00 00 00 62 7B 84 00 00 04 04              .....b{.....
```

- The *length in bits* field (bytes 07–0AH) starts with a byte containing 84H, which indicates that the actual size of the variable is represented as a 3-byte value (the following 3 bytes). Bytes 08–0AH contain the value 040000H, the size of the 32 KB array in bits.

This third C statement, because it declares a FAR variable, causes two TYPDEF records to be generated:

```
char    far     foo3[10][2][20];        /* 400-element FAR array */
```

The two TYPDEF records are

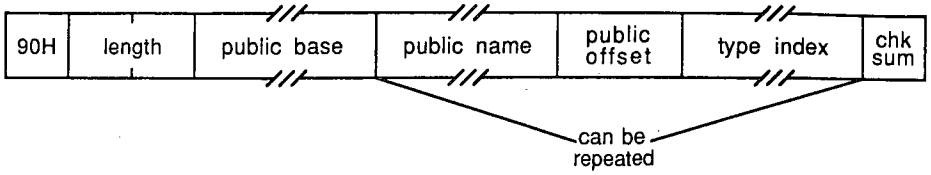
```
       0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  8E 06 00 00 00 62 7B 08 87 8E 09 00 00 00 61 77  .....b{.......aw
0010  81 90 01 01 7E                                    ....~
```

- Bytes 00–08H contain the first TYPDEF record, which defines the data type of the elements of the array (NEAR, scalar, 8 bits in size).
- Bytes 09–14H contain the second TYPDEF record. The *leaf descriptor* field of this record declares that the variable is FAR (byte 0EH contains 61H) and an array (byte 0FH, the *variable type*, contains 77H).
  - Because this TYPDEF record describes a FAR variable, bytes 10–12H represent a *number of elements* field. The first byte of the field is 81H, indicating a 2-byte value, so the next 2 bytes (bytes 11–12H) contain the number of elements in the array, 0190H (400D).
- Byte 13H (the *element type index*) contains 01H, which is a reference to the first TYPDEF record in the object module — in this example, the one in bytes 00–08H.

90H PUBDEF Public Names Definition Record

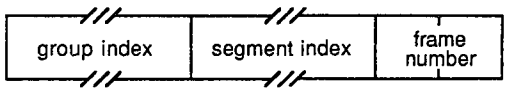
The PUBDEF record contains a list of public names. When object modules are linked, the linker uses these names to resolve external references in other object modules.

Record format



Public base

Each name in the PUBDEF record refers to a location (a 16-bit offset) in a particular segment or group. The *public base*, a variable-length field that specifies the segment or group, is formatted as follows:



- *Group index* is an index field that references a previous GRPDEF record in the object module. If the *group index* value is 0, no group is associated with this PUBDEF record.
- *Segment index* is also an index field. It associates a particular segment with this PUBDEF record by referencing a previous SEGDEF record. A value of 1 indicates the first SEGDEF record in the object module, a value of 2 indicates the second, and so on. If the *segment index* value is 0, the *group index* must also be 0—in this case, the *frame number* appears in the *public base* field.
- The 2-byte *frame number* appears in the *public base* field only when the *group index* and *segment index* are both 0. In other words, the *frame number* specifies the start of an absolute segment. If present, the value in the *frame number* field indicates the number of the frame containing the public name.

Public name

Public name is a variable-length field containing a public name. The first byte specifies the length of the name; the remainder is the name itself. (The Intel specification allows names of 1 to 255 bytes. Microsoft LINK restricts the maximum length of a public name to 127 bytes.)

Public offset

Public offset is a 2-byte field containing the offset of the location referred to by the *public name*. This offset is assumed to lie within the segment, group, or frame specified in the *public base* field.

Type index

Type index is an index field that references a previous TYPDEF record in the object module. A value of 1 indicates the first TYPDEF record in the module, a value of 2 indicates the second, and so on. The *type index* value can be 0 if no data type is associated with the public name.

The *public name*, *public offset*, and *type index* fields can be repeated within a single PUBDEF record. Thus, one PUBDEF record can declare a list of public names.

Location in object module

Any PUBDEF records in an object module must appear after the GRPDEF and SEGDEF records to which they refer. Because PUBDEF records are not themselves referenced by any other type of object record, they are generally placed near the end of an object module.

Examples

The following two examples show PUBDEF records created by the Microsoft Macro Assembler.

The first example is the record for the statement

```
PUBLIC  GAMMA
```

The PUBDEF record is

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  90 0C 00 00 01 05 47 41 4D 4D 41 02 00 00 F9     ......GAMMA....
```

- Byte 00H contains 90H, indicating a PUBDEF record.
- Bytes 01–02H contain 000CH, the length of the remainder of the record.
- Bytes 03–04H represent the *public base* field. Byte 03H (the *group index*) contains 0, indicating that no group is associated with the name in this PUBDEF record. Byte 04H (the *segment index*) contains 1, a reference to the first SEGDEF record in the object module. This is the segment to which the name in this PUBDEF record refers.
- Bytes 05–0AH represent the *public name* field. Byte 05H contains 05H (the length of the name), and bytes 06–0AH contain the name itself, *GAMMA*.
- Bytes 0B–0CH contain 0002H, the *public offset*. The name *GAMMA* thus refers to the location that is offset 2 bytes from the beginning of the segment referenced by the *public base*.
- Byte 0DH is the *type index*. The value of the *type index* is 0, indicating that no data type is associated with the name *GAMMA*.
- Byte 0EH contains the checksum, 0F9H.

The next example is the PUBDEF record for the following absolute symbol declaration:

```
        PUBLIC    ALPHA
ALPHA   EQU       1234h
```

The PUBDEF record is

```
       0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000   90 0E 00 00 00 00 00 05 41 4C 50 48 41 34 12 00  ......ALPHA4....
0010   B1
```

- Bytes 03–06H (the *public base* field) contain a *group index* of 0 (byte 03H) and a *segment index* of 0 (byte 04H). Since both the *group index* and *segment index* are 0, a *frame number* also appears in the *public base* field. In this instance, the *frame number* (bytes 05–06H) also happens to be 0.
- Bytes 07–0CH (the *public name* field) contain the name *ALPHA*, preceded by its length.
- Bytes 0D–0EH (the *public offset* field) contain 1234H. This is the value associated with the symbol *ALPHA* in the assembler EQU directive. If *ALPHA* is declared in another object module with the declaration

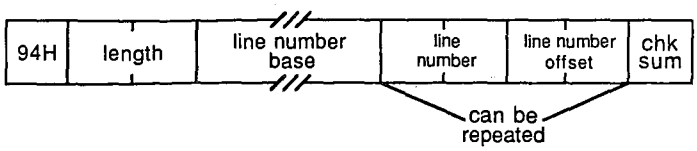
  ```
  EXTRN     ALPHA:ABS
  ```

  any references to *ALPHA* in that object module are fixed up as absolute references to offset 1234H in frame 0. In other words, *ALPHA* would have the value 1234H.
- Byte 0FH (the *type index*) contains 0.

94H LINNUM Line Number Record

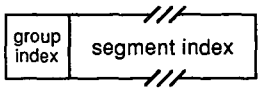
The LINNUM record relates line numbers in source code to addresses in object code.

Record format



Line number base

The *line number base* describes the segment to which the line number refers. Although the complete Intel specification allows the line number base to refer to a group or to an absolute segment as well as to a relocatable segment, Microsoft restricts references in this field to relocatable segments. The format of the *line number base* field is



- The *group index* field always contains a single zero byte.
- The *segment index* is an index field that references a previous SEGDEF record. A value of 1 indicates the first SEGDEF record in the object module, a value of 2 indicates the second, and so on.

Line number

Line number is a 2-byte field containing a line number between 0 and 32,767 (0–7FFFH).

Line number offset

The *line number offset* is a 2-byte field that specifies the offset of the executable code (in the segment specified in the *line number base* field) to which the line number in the *line number* field refers.

The *line number* and *line number offset* fields can be repeated, so a single LINNUM record can specify multiple line numbers in the same segment.

Location in object module

Any LINNUM records in an object module must appear after the SEGDEF records to which they refer. Because LINNUM records are not themselves referenced by any other type of object record, they are generally placed near the end of an object module.

Example

The following LINNUM record was generated by the Microsoft C Compiler:

```
       0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000   94 0F 00 00 01 02 00 00 00 03 00 08 00 04 00 0F  ................
0010   00 3C                                           ..
```

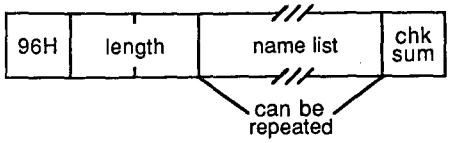
- Byte 00H contains 94H, indicating that this is a LINNUM record.
- Bytes 01–02H contain 000FH, the length of the remainder of the record.
- Bytes 03–04H represent the *line number base* field. Byte 03H (the *group index* field) contains 00H, as it must. Byte 04H (the *segment index* field) contains 01H, indicating that the line numbers in this LINNUM record refer to code in the segment defined in the first SEGDEF record in this object module.
- Bytes 05–06H (a *line number* field) contain 0002H, and bytes 07–08H (a *line number offset* field) contain 0000H. Together, they indicate that source-code line number 0002 corresponds to offset 0000H in the segment indicated in the *line number base* field.

  Similarly, the two pairs of *line number* and *line number offset* fields in bytes 09–10H specify that line number 0003 corresponds to offset 0008H and that line number 0004 corresponds to offset 000FH.
- Byte 11H contains the checksum, 3CH.

96H LNAMES List of Names Record

The LNAMES record is a list of names that can be referenced by subsequent SEGDEF and GRPDEF records in the object module.

Record format



Name list

Name list is a variable-length field that contains the list of names. Each name is preceded by 1 byte that defines its length, which can be a value between 0 and 255 (0–0FFH).

The names in the list are indexed implicitly in the order they appear: The first name in the list has an index of 1, the second name has an index of 2, and so forth. References to the names contained in *name list* by subsequent object records, such as SEGDEF, are accomplished by using this index number. LINK imposes a limit of 255 logical names per object module.

Location in object module

Any LNAMES records in an object module must appear before the GRPDEF or SEGDEF records that refer to them. Because it does not refer to any other type of object records, an LNAMES record usually appears near the start of an object module.

Example

The following LNAMES record contains the segment and class names specified in all three of the assembler statements:

```
_TEXT    SEGMENT byte public 'CODE'
_DATA    SEGMENT word public 'DATA'
_STACK   SEGMENT para public 'STACK'
```

The LNAMES record is

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  96 25 00 00 04 43 4F 44 45 04 44 41 54 41 05 53  .%...CODE.DATA.S
0010  54 41 43 4B 05 5F 44 41 54 41 06 5F 53 54 41 43  TACK._DATA._STAC
0020  4B 05 5F 54 45 58 54 8B                          K._TEXT.
```

- Byte 00H contains 96H, indicating that this is an LNAMES record.
- Bytes 01–02H contain 0025H, the length of the remainder of the record.

- Byte 03H contains 00H, a zero-length name.
- Byte 04H contains 04H, the length of the class name CODE, which is found in bytes 05–08H. Bytes 09–26H contain the class names DATA and STACK and the segment names *_DATA*, *_STACK*, and *_TEXT*, each preceded by 1 byte giving its length.
- Byte 27H contains the checksum, 8BH.

98H SEGDEF Segment Definition Record

The SEGDEF record describes a logical segment in an object module. It defines the segment's name, length, and alignment, and the way the segment can be combined with other logical segments. LINK imposes a limit of 255 SEGDEF records per object module.

Object records that follow a SEGDEF record can refer to it to identify a particular segment.

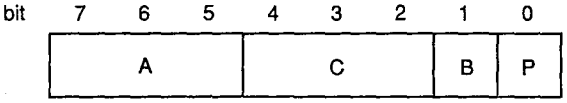
Record format

Segment attributes

Segment attributes is a variable-length field:

The ACBP byte

The contents and size of the *segment attributes* field depend on the first byte of the field, the ACBP byte:



The bit fields in the ACBP byte describe the following characteristics of the segment:

A **A**lignment in the run-time memory map
C **C**ombination with other segments
B **B**ig (a segment of exactly 64 KB)
P **P**age-resident (not used in MS-DOS)

The A field. Bits 7–5 of the ACBP byte, the *A* field, describe the logical segment's alignment:

A = 0 (000B) Absolute (located at a specified frame address)
A = 1 (001B) Relocatable, byte aligned
A = 2 (010B) Relocatable, word aligned
A = 3 (011B) Relocatable, paragraph aligned
A = 4 (100B) Relocatable, page aligned

The original Intel specification includes two additional segment-alignment values not supported in MS-DOS.

The following examples of Microsoft assembler SEGMENT directives show the resulting values for the *A* field in the corresponding SEGDEF object record:

```
aseg    SEGMENT at 400h                 ; A = 0
bseg    SEGMENT byte public 'CODE'      ; A = 1
cseg    SEGMENT para stack 'STACK'      ; A = 3
```

The C field. Bits 4–2 of the ACBP byte, the *C* field, describe how the linker can combine the segment with other segments. Under MS-DOS, segments with the same name and class can be combined in two ways. They can be concatenated to form one logical segment, or they can be overlapped. In the latter case, they have either the same starting address or the same end address and they describe a common area of memory.

The value in the *C* field corresponds to one of these two methods of combining segments. Meaningful values, however, also depend on whether the segment is absolute (A = 0) or relocatable (A = 1, 2, 3, or 4). If A = 0, then *C* must also be 0, because absolute segments cannot be combined. Values for the *C* field are

C = 0 (000B)	Cannot be combined; used for segments whose combine type is not explicitly specified (private segments).
C = 1 (001B)	Not used by Microsoft.
C = 2 (010B)	Can be concatenated with another segment of the same name; used for segments with the *public* combine type.
C = 3 (011B)	Undefined.
C = 4 (100B)	As defined by Microsoft, same as C = 2.
C = 5 (101B)	Can be concatenated with another segment with the same name; used for segments with the *stack* combine type.
C = 6 (110B)	Can be overlapped with another segment with the same name; used for segments with the *common* combine type.
C = 7 (111B)	As defined by Microsoft, same as C = 2.

The following examples of assembler SEGMENT directives show the resulting values for the *C* field in the corresponding SEGDEF object record:

```
aseg    SEGMENT at 400H             ; C = 0
bseg    SEGMENT public 'DATA'       ; C = 2
cseg    SEGMENT stack 'STACK'       ; C = 5
dseg    SEGMENT common 'COMMON'     ; C = 6
```

See PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING TOOLS: The Microsoft Object Linker.

The B and P fields. Bit 1 of the ACBP byte, the *B* field, is set to 1 (and the *segment length* field is set to 0) only if the segment is exactly 64 KB long.

Bit 0 of the ACBP byte, the *P* field, is unused in MS-DOS. Its value should always be 0.

Frame number and offset

The *frame number* and *offset* fields of the *segment attributes* field are present only if the segment is an absolute segment (A = 0 in the ACBP byte). Taken together, the *frame number* and *offset* indicate the starting address of the segment.

- *Frame number* is a 2-byte field that contains the frame number of the start of the segment.
- *Offset* is a 1-byte field that contains an offset between 00H and 0FH within the specified frame. LINK ignores the *offset* field.

Segment length

Segment length is a 2-byte field that specifies the length of the segment in bytes. The length can be from 00H to FFFFH. If a segment is exactly 64 KB (10000H) in size, *segment length* should be 0 and the *B* field in the ACBP byte should be 1.

Segment name index, class name index, and overlay name index

Each of the *segment name index*, *class name index*, and *overlay name index* fields contains an index into the list of names defined in previous LNAMES records in the object module. An index value of 1 indicates the first name in the LNAMES record, a value of 2 the second, and so on.

- The *segment name index* identifies the segment with a unique name. The name may have been assigned by the programmer, or it may have been generated by a compiler.
- The *class name index* identifies the segment with a class name (such as CODE, FAR_DATA, and STACK). The linker places segments with the same class name into a contiguous area of memory in the run-time memory map.
- The *overlay name index* identifies the segment with a run-time overlay. Starting with version 2.40, however, LINK ignores the *overlay name index*. In versions 2.40 and later, command-line parameters to LINK, rather than information contained in object modules, determine the creation of run-time overlays.

Location in object module

SEGDEF records must follow the LNAMES record to which they refer. In addition, SEGDEF records must precede any PUBDEF, LINNUM, GRPDEF, FIXUPP, LEDATA, or LIDATA records that refer to them.

Examples

In this first example, the segment is byte aligned:

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  98 07 00 28 11 00 07 02 01 1E                  ...(......
```

- Byte 00H contains 98H, indicating that this is a SEGDEF record.
- Bytes 01–02H contain 0007H, the length of the remainder of the record.

- Byte 03H contains 28H (00101000B), the ACBP byte. Bits 7–5 (the *A* field) contain 1 (001B), indicating that this segment is relocatable and byte aligned. Bits 4–2 (the *C* field) contain 2 (010B), which represents a *public* combine type. (When this object module is linked, this segment will be concatenated with all other segments with the same name.) Bit 1 (the *B* field) is 0, indicating that this segment is smaller than 64 KB. Bit 0 (the *P* field) is ignored and should be zero, as it is here.
- Bytes 04–05H contain 0011H, the size of the segment in bytes.
- Bytes 06–08H index the list of names defined in the module's LNAMES record. Byte 06H (*the segment name index*) contains 07H, so the name of this segment is the seventh name in the LNAMES record. Byte 07H (the *class name index*) contains 02H, so the segment's class name is the second name in the LNAMES record. Byte 08H (the *overlay name index*) contains 1, a reference to the first name in the LNAMES record. (This name is usually null, as MS-DOS ignores it anyway.)
- Byte 09H contains the checksum, 1EH.

The second SEGDEF record declares a word-aligned segment. It differs only slightly from the first.

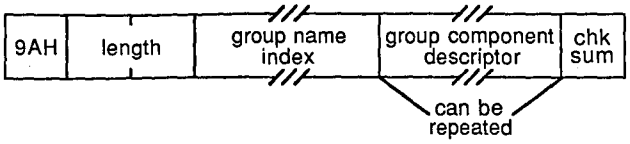
```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  98 07 00 48 0F 00 05 03 01 01                  ...H......
```

- Bits 7–5 (the *A* field) of byte 03H (the ACBP byte) contain 2 (010B), indicating that this segment is relocatable and word aligned.
- Bytes 04–05H contain the size of the segment, 000FH.
- Byte 06H (the *segment name index*) contains 05H, which refers to the fifth name in the previous LNAMES record.
- Byte 07H (the *class name index*) contains 03H, a reference to the third name in the LNAMES record.

9AH GRPDEF Group Definition Record

The GRPDEF record defines a group of segments, all of which lie within the same 64 KB frame in the run-time memory map. LINK imposes a limit of 21 GRPDEF records per object module.

Record format

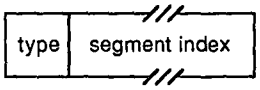


Group name index

Group name index is an index field whose value refers to a name in the *name list* field of a previous LNAMES record.

Group component descriptor

The *group component descriptor* consists of two fields:



- *Type* is a 1-byte field whose value is always 0FFH, indicating that the following field contains a *segment index* value. The original Intel specification defines four other types of *group component descriptor* with the values 0FEH, 0FDH, 0FBH, and 0FAH. LINK ignores these other *type* values, however, and assumes that the *group component descriptor* contains a *segment index* value.
- The *segment index* field contains an index number that refers to a previous SEGDEF record. A value of 1 indicates the first SEGDEF record in the object module, a value of 2 indicates the second, and so on.

The *group component descriptor* field is usually repeated within the GRPDEF record, so all segments constituting the group can be included in one GRPDEF record.

Location in object module

GRPDEF records must follow the LNAMES and SEGDEF records to which they refer. They must also precede any PUBDEF, LINNUM, FIXUPP, LEDATA, or LIDATA records that refer to them.

Example

The following example of a GRPDEF record corresponds to the assembler directive:

```
tgroup  GROUP seg1,seg2,seg3
```

The GRPDEF record is

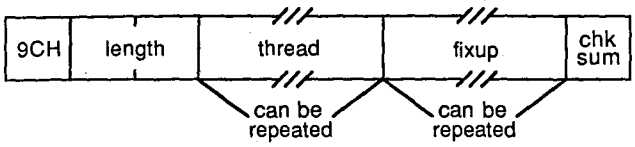
```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  9A 08 00 06 FF 01 FF 02 FF 03 55                ..........U
```

- Byte 00H contains 9AH, indicating that this is a GRPDEF record.
- Bytes 01–02H contain 0008H, the length of the remainder of the record.
- Byte 03H contains 06H, the *group name index*. In this instance, the index number refers to the sixth name in the previous LNAMES record in the object module. That name is the name of the group of segments defined in the remainder of the record.
- Bytes 04–05H contain the first of three *group component descriptor* fields. Byte 04H contains the required 0FFH, indicating that the subsequent field is a *segment index*. Byte 05H contains 01H, a *segment index* that refers to the first SEGDEF record in the object module. This SEGDEF record declared the first of three segments in the group.
- Bytes 06–07H represent the second *group component descriptor*, this one referring to the second SEGDEF record in the object module.
- Similarly, bytes 08–09H are a *group component descriptor* field that references the third SEGDEF record.
- Byte 0AH contains the checksum, 55H.

9CH FIXUPP Fixup Record

The FIXUPP record contains information that allows the linker to resolve (fix up) addresses whose values cannot be determined by the language translator. FIXUPP records describe the LOCATION of each address value to be fixed up, the TARGET address to which the fixup refers, and the FRAME relative to which the address computation is performed.

Record format



Thread and fixup fields

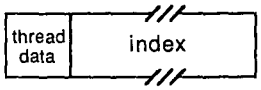
A FIXUPP record can contain zero or more *thread* fields and zero or more *fixup* fields. Each *fixup* field describes the method to be used by the linker to compute the TARGET address to be placed at a particular location in the executable image, relative to a particular FRAME. The information that determines the LOCATION, TARGET, and FRAME can be specified explicitly in the *fixup* field. It can also be specified within the *fixup* field by a reference to a previous *thread* field.

A *thread* field describes only the method to be used by the linker to refer to a particular TARGET or FRAME. Because the same *thread* field can be referenced in several subsequent *fixup* fields, a FIXUPP record that uses *thread* fields may be smaller than one in which *thread* fields are not used.

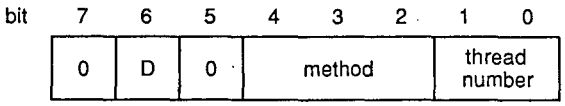
Thread and *fixup* fields are distinguished from one another by the high-order bit of the first byte in the field. If the high-order bit is 0, the field is a *thread* field. If the high-order bit is 1, the field is a *fixup* field.

The thread field

A *thread* field contains information that can be referenced in subsequent *thread* or *fixup* fields in the same or subsequent FIXUPP records. It has the following format:



The *thread data* field is a single byte comprising five subfields:



- Bit 7 of the *thread data* byte is 0, indicating the start of a *thread* field.
- The *D* field (bit 6) indicates whether the *thread* field specifies a FRAME or a TARGET. The *D* bit is set to 1 to indicate a FRAME or to 0 to indicate a TARGET.
- Bit 5 of the *thread data* byte is not used. It should always be set to 0.
- Bits 4 through 2 represent the *method* field. If *D* = 1, the *method* field contains 0, 1, 2, 4, or 5. Each of these numbers corresponds to one method of specifying a FRAME (*see* Table 19-2). If *D* = 0, the *method* field contains 0, 1, 2, 4, 5, or 6, each of which corresponds to one of the methods of specifying a TARGET (*see* Table 19-3).

  In the case of a TARGET address, only bits 3 and 2 of the *method* field are used. When *D* = 0, the high-order bit of the value in the *method* field is derived from the *P* bit in the *fix dat* field of any subsequent *fixup* field that refers to this *thread* field. Thus, if *D* = 0, bit 4 of the *method* field is also 0, and the only meaningful values for the *method* field are 0, 1, and 2.
- The *thread number* field (bits 1 and 0) contains a number between 0 and 3. This number is used in subsequent *fixup* or *thread* fields to refer to this particular *thread* field.

  The *thread number* is implicitly associated with the *D* field by the linker, so as many as eight different *thread* fields (four FRAMEs and four TARGETs) can be referenced at any time. A *thread number* can be reused in an object module and, if it is, always refers to the *thread* field in which it last appeared.

Table 19-2. FRAME Fixup Methods.

Method	Description
0	The FRAME is specified by a segment index.
1	The FRAME is specified by a group index.
2	The FRAME is indicated by an external index. LINK determines the FRAME from the external name's corresponding PUBDEF record in another object module, which specifies either a logical segment or a group.
3	The FRAME is identified by an explicit frame number. (Not supported by LINK.)
4	The FRAME is determined by the segment in which the LOCATION is defined. In this case, the largest possible frame number is used.
5	The FRAME is determined by the TARGET's segment, group, or external index.

Table 19-3. TARGET Fixup Methods.

Method	Description
0	The TARGET is specified by a segment index and a displacement. The displacement is given in the *target displacement* field of the FIXUPP record.
1	The TARGET is specified by a group index and a *target displacement.*
2	The TARGET is specified by an external index and a *target displacement.* LINK adds the displacement to the address it determines from the external name's corresponding PUBDEF record in another object module.
3	The TARGET is identified by an explicit frame number. (Not supported by LINK.)
4*	The TARGET is specified by a segment index only.
5*	The TARGET is specified by a group index only.
6*	The TARGET is specified by an external index. The TARGET is the address associated with the external name.
7*	The TARGET is identified by an explicit frame number. (Not supported by LINK.)

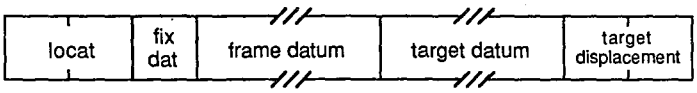
*TARGET methods 4–7 are analogous to the preceding four, except that methods 4–7 do not use an explicit displacement to identify the TARGET. Instead, a displacement of 0 is assumed.

The *index* field either contains an index value that refers to a previous SEGDEF, GRPDEF, or EXTDEF record, or it contains an explicit frame number. The interpretation of the index value depends on the value of the *method* field of the *thread data* field:

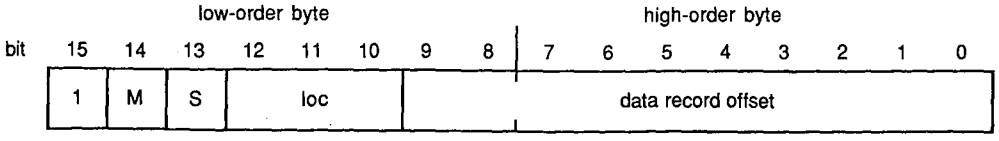
method = 0	Segment index (reference to a previous SEGDEF record)
method = 1	Group index (reference to a previous GRPDEF record)
method = 2	External index (reference to a previous EXTDEF record)
method = 3	Frame number (not supported by LINK; ignored)

The fixup field

The *fixup* field provides the information needed by the linker to resolve a reference to a relocatable or external address. The *fixup* field has the following format:



The 2-byte *locat* field has an unusual format. Contrary to the usual byte order in Intel data structures, the most significant bits of the *locat* field are found in the low-order, rather than the high-order, byte:

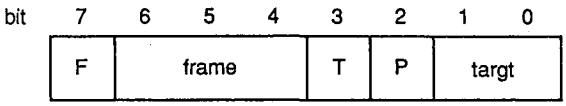


- Bit 15 (the high-order bit of the *locat* field) contains 1, indicating that this is a *fixup* field.
- Bit 14 (the *M* bit) is 1 if the fixup is segment relative and 0 if the fixup is self-relative.
- Bit 13 (the *S* bit) is currently unused and should always be set to 0.
- Bits 12 through 10 represent the *loc* field. This field contains a number between 0 and 5 that indicates the type of LOCATION to be fixed up:

loc = 0	Low-order byte
loc = 1	Offset
loc = 2	Segment
loc = 3	Pointer (segment:offset)
loc = 4	High-order byte (not recognized by LINK)
loc = 5	Loader-resolved offset (treated as *loc* = 1 by the linker)

- Bits 9 through 0 (the *data record offset*) indicate the position of the LOCATION to be fixed up in the LEDATA or LIDATA record immediately preceding the FIXUPP record. This offset indicates either a byte in the *data* field of an LEDATA record or a data byte in the *content* field of an *iterated data block* in an LIDATA record.

The *fix dat* field is a single byte comprising five fields:



- Bit 7 (the *F* bit) is set to 1 if the FRAME for this fixup is specified by a reference to a previous *thread* field. The *F* bit is 0 if the FRAME method is explicitly defined in this *fixup* field.
- The interpretation of the *frame* field in bits 6 through 4 depends on the value of the *F* bit. If *F* = 1, the *frame* field contains a number between 0 and 3 that indicates the *thread* field containing the FRAME method. If *F* = 0, the *frame* field contains 0, 1, 2, 4, or 5, corresponding to one of the methods of specifying a FRAME listed in Table 19-2.
- Bit 3 (the *T* bit) is set to 1 if the TARGET for the fixup is specified by a reference to a previous *thread* field. If the *T* bit is 0, the TARGET is explicitly defined in this *fixup* field.
- Bit 2 (the *P* bit) and bits 1 and 0 (the *targt* field) can be considered a 3-bit field analogous to the *frame* field.
- If the *T* bit indicates that the TARGET is specified by a previous *thread* reference (*T* = 1), the *targt* field contains a number between 0 and 3 that refers to a previous *thread* field containing the TARGET method. In this case, the *P* bit, combined with the 2 low-order bits of the *method* field in the *thread* field, determines the TARGET method.

If the *T* bit is 0, indicating that the target is explicitly defined, the *P* and *targt* fields together contain 0, 1, 2, 4, 5, or 6. This number corresponds to one of the TARGET fixup methods listed in Table 19-3. (In this case, the *P* bit can be regarded as the high-order bit of the method number.)

Frame datum is an index field that refers to a previous SEGDEF, GRPDEF, or EXTDEF record, depending on the FRAME method.

Similarly, the *target datum* field contains a segment index, a group index, or an external index, depending on the TARGET method.

The *target displacement* field, a 2-byte field, is present only if the *P* bit in the *fixdat* field is set to 0, in which case the *target displacement* field contains the 16-bit offset used in methods 0, 1, and 2 of specifying a TARGET.

Location in object module

FIXUPP records must appear after the SEGDEF, GRPDEF, or EXTDEF records to which they refer. In addition, if a FIXUPP record contains any *fixup* fields, it must immediately follow the LEDATA or LIDATA record to which the fixups refer.

Examples

Although crucial to the proper linking of object modules, FIXUPP records are terse: Almost every bit is meaningful. For these reasons, the following three examples of FIXUPP records are particularly detailed.

A good way to understand how a FIXUPP record is put together is to compare it to the corresponding source code. The Microsoft Macro Assembler is helpful in this regard, because it marks in its source listing address references it cannot resolve. The "program" in Figure 19-6 is designed to show how some of the most frequently encountered fixups are encoded in FIXUPP records.

```
                                TITLE   fixupps
                        _TEXT   SEGMENT byte public 'CODE'
                                ASSUME  cs:_TEXT
                                EXTRN   NearLabel:near
                                EXTRN   FarLabel:far

0000                    NearProc        PROC    near

0000  E9 0000 E                        jmp     NearLabel       ;relocatable word offset
0003  EB 00 E                          jmp     short NearLabel ;relocatable byte offset
0005  EA 0000 ---- R                    jmp     far ptr FarProc ;far jump to a known seg
000A  EA 0000 ---- E                    jmp     FarLabel     ;far jump to an unknown seg

000F  BB 0015 R                         mov     bx,offset LocalLabel ;relocatable offset
0012  B8 ---- R                         mov     ax,seg LocalLabel    ;relocatable seg
```

Figure 19-6. A sample "program" showing how some common fixups are encoded in FIXUPP records. (more)

```
0015  C3                    LocalLabel:     ret

                            NearProc        ENDP

0016                        _TEXT   ENDS


0000                        FAR_TEXT        SEGMENT byte public 'FAR_CODE'
                                    ASSUME  cs:FAR_TEXT

0000                        FarProc PROC    far

0000  CB                            ret

                            FarProc ENDP

0001                        FAR_TEXT        ENDS

                                    END
```

Figure 19-6. Continued.

The assembler generates one LEDATA record for this program:

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0010  A0 1A 00 01 00 00 E9 00 00 EB 00 EA 00 00 00 00  ................
0020  EA 00 00 00 00 BB 00 00 B8 00 00 C3 67           ............g
```

Bytes 06–2BH (the *data* field) of this LEDATA record contain 8086 opcodes for each of the instruction mnemonics in the source code. The gaps (zero values) in the *data* field correspond to address values that the assembler cannot resolve. The linker will fix up the address values in the gaps by computing the correct values and adding them to the zero values in the gaps. The FIXUPP record that tells the linker how to do this immediately follows the LEDATA record in the object module:

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  9C 21 00 84 01 06 01 02 80 04 06 01 02 CC 06 04  .!..............
0010  02 02 CC 0B 06 01 01 C4 10 00 01 01 15 00 C8 13  ................
0020  04 01 01 A3                                      ....
```

- Byte 00H contains 9CH, indicating this is a FIXUPP record.
- Bytes 01–02H contain 0021H, the length of the remainder of the record.
- Bytes 03–07H represent the first of the six *fixup* fields in this record:

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  9C 21 00 84 01 06 01 02 80 04 06 01 02 CC 06 04  .!..............
0010  02 02 CC 0B 06 01 01 C4 10 00 01 01 15 00 C8 13  ................
0020  04 01 01 A3                                      ....
```

The information in this *fixup* field will allow the linker to resolve the address reference in the statement

```
jmp     NearLabel
```

- Bytes 03–04H (the *locat* field) contain 8401H (1000010000000001B). (Recall that this field does not conform to the usual Intel byte order.) Bit 15 is 1, signifying that this is a *fixup* field, not a *thread* field. Bit 14 (the *M* bit) is 0, so this fixup is self-relative. Bit 13 is unused and should be set to 0, as it is here. Bits 12–10 (the *loc* field) contain 1 (001B), so the LOCATION to be fixed up is a 16-bit offset. Bits 9–0 (the *data record offset*) contain 1 (0000000001B), which informs the linker that the LOCATION to be fixed up is at offset 1 in the *data* field of the LEDATA record immediately preceding this FIXUPP record—in other words, the 2 bytes immediately following the first opcode 0E9H.
- Byte 05H (the *fix dat* field) contains 06H (00000110B). Bit 7 (the *F* bit) is 0, meaning the FRAME for this fixup is explicitly specified in this *fixup* field. Bits 6–4 (the *frame* field) contain 0 (000B), indicating that FRAME method 0 specifies the FRAME. Bit 3 (the *T* bit) is 0, so the TARGET for this fixup is also explicitly specified. Bits 2–0 (the *P* bit) and the *targt* field contain 6 (110B), so TARGET method 6 specifies the TARGET.
- Byte 06H is a *frame datum* field, because the FRAME is explicitly specified (the *F* bit of the *fix dat* field = 0). And, because method 0 is specified, the *frame datum* is an index field that refers to a previous SEGDEF record. In this example, the *frame datum* field contains 1, which indicates the first SEGDEF record in the object module: the _TEXT segment.
- Similarly, byte 07H is a *target datum*, because the TARGET is also explicitly specified (the *T* bit of the *fix dat* field = 0). The *fix dat* field also indicates that TARGET method 6 is used, so the *target datum* is an index field that refers to the *external reference list* in a previous EXTDEF record. The value of this index is 2, so the TARGET is the second external reference declared in the EXTDEF record: *NearLabel* in this object module.

- Bytes 08–0CH represent the second *fixup* field:

```
        0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  9C 21 00 84 01 06 01 02 80 04 06 01 02 CC 06 04  .!..............
0010  02 02 CC 0B 06 01 01 C4 10 00 01 01 15 00 C8 13  ................
0020  04 01 01 A3                                      ....
```

This *fixup* field corresponds to the statement

```
jmp     short NearLabel
```

The only difference between this statement and the first is that the jump uses an 8-bit, rather than a 16-bit, offset. Thus, the *loc* field (bits 12–10 of byte 08H) contains 0 (000B) to indicate that the LOCATION to be fixed up is a low-order byte.

- Bytes 0D–11H represent the third *fixup* field in this FIXUPP record:

```
        0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  9C 21 00 84 01 06 01 02 80 04 06 01 02 CC 06 04  .!..............
0010  02 02 CC 0B 06 01 01 C4 10 00 01 01 15 00 C8 13  ................
0020  04 01 01 A3                                      ....
```

This *fixup* field corresponds to the statement

```
jmp     far ptr FarProc
```

In this case, both the TARGET's frame (the segment *FAR_TEXT*) and offset (the label *FarProc*) are known to the assembler. Both the segment address and the label offset are relocatable, however, so in the FIXUPP record the assembler passes the responsibility for resolving the addresses to the linker.

- Bytes 0D–0EH (the *locat* field) indicate that the field is a *fixup* field (bit 15 = 1) and that the fixup is segment relative (bit 14—the *M* bit = 1). The *loc* field (bits 12–10) contains 3 (011B), so the LOCATION being fixed up is a 32-bit (FAR) pointer (segment and offset). The *data record offset* (bits 9–0) is 6 (0000000110B); the LOCATION is the 4 bytes following the first far jump opcode (EAH) in the preceding LEDATA record.
- In byte 0FH (the *fix dat* field), the *F* bit and the *frame* field are 0, indicating that method 0 (a segment index) is used to specify the FRAME. The *T* bit is 0 (meaning the *target* is explicitly defined in the *fixup* field); therefore, the *P* bit and *targt* fields together indicate method 4 (a segment index) to specify the TARGET.
- Because the FRAME is specified with a segment index, byte 10H (the *frame datum* field) is a reference to the second SEGDEF record in the object module, which in this example declared the *FAR_TEXT* segment. Similarly, byte 11H (the *target datum* field) references the *FAR_TEXT* segment. In this case, the FRAME is the same as the TARGET segment; had *FAR_TEXT* been one of a group of segments, the FRAME could have referred to the group instead.

- The fourth assembler statement is different from the third because it references a segment not known to the assembler:

```
jmp     FarLabel
```

Bytes 12–16H contain the corresponding *fixup* field:

```
          0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000   9C 21 00 84 01 06 01 02 80 04 06 01 02 CC 06 04   .!..............
0010   02 02 CC 0B 06 01 01 C4 10 00 01 01 15 00 C8 13   ................
0020   04 01 01 A3                                       ....
```

The significant difference between this and the preceding *fixup* field is that the *P* bit and *targt* field of the *fix dat* byte (byte 14H) specify TARGET method 6. In this *fixup* field, the *target datum* (byte 16H) refers to the first EXTDEF record in the object module, which declares *FarLabel* as an external reference.

- The fifth *fixup* field (bytes 17–1DH) is

```
          0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000   9C 21 00 84 01 06 01 02 80 04 06 01 02 CC 06 04   .!..............
0010   02 02 CC 0B 06 01 01 C4 10 00 01 01 15 00 C8 13   ................
0020   04 01 01 A3                                       ....
```

This *fixup* field contains information that enables the linker to calculate the value of the relocatable offset *LocalLabel*:

```
mov     bx,offset LocalLabel
```

- Bytes 17–18H (the *locat* field) contain C410H (1100010000010000B). Bit 15 is 1, denoting a *fixup* field. The *M* bit (bit 14) is 1, indicating that this fixup is segment relative. The *loc* field (bits 12–10) contains 1 (001B), so the LOCATION is a 16-bit offset. The *data record offset* (bits 9–0) is 10H (0000010000B), a reference to the 2 bytes in the LEDATA record following the opcode 0BBH.
- Byte 19H (the *fix dat* byte) contains 00H. The *F* bit, *frame* field, *T* bit, *P* bit, and *targt* field are all 0, so FRAME method 0 and TARGET method 0 are explicitly specified in this *fixup* field.
- Because FRAME method 0 is used, byte 1AH (the *frame datum* field) is an index field. It contains 01H, a reference to the first SEGDEF record in the object module, which declares the segment *_TEXT*.

  Similarly, byte 1BH (the *target datum* field) references the *_TEXT* segment.
- Because TARGET method 0 is specified, an offset, in addition to a segment, is required to define the TARGET. This offset appears in the *target displacement* field in bytes 1C–1DH. The value of this offset is 0015H, corresponding to the offset of the TARGET (*LocalLabel*) in its segment (*_TEXT*).

- The sixth and final *fixup* field in this FIXUPP record (bytes 1E–22H) is

```
        0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  9C 21 00 84 01 06 01 02 80 04 06 01 02 CC 06 04  .!..............
0010  02 02 CC 0B 06 01 01 C4 10 00 01 01 15 00 C8 13  ................
0020  04 01 01 A3                                      ....
```

This corresponds to the segment of the relocatable address *LocalLabel*:

```
mov     ax,seg LocalLabel
```

- Bytes 1E–1FH (the *locat* field) contain C813H (1100100000010011B). Bit 15 is 1, so this is a *fixup* field. The *M* bit (bit 14) is 1, so the fixup is segment relative. The *loc* field (bits 12–10) contains 2 (010B), so the LOCATION is a 16-bit segment value. The *data record offset* (bits 9–0) indicates the 2 bytes in the LEDATA record following the opcode 0B8H.
- Byte 20H (the *fix dat* byte) contains 04H, so FRAME method 0 and TARGET method 4 are explicitly specified in this *fixup* field.
- Byte 21H (the *frame datum* field) contains 01H. Because FRAME method 0 is specified, the *frame datum* is an index value that refers to the first SEGDEF record in the object module (corresponding to the *_TEXT* segment).
- Byte 22H (the *target datum* field) contains 01H. Because TARGET method 4 is specified, the *target datum* also references the *_TEXT* segment.

- Finally, byte 23H contains this FIXUPP record's checksum, 0A3H.

The next two FIXUPP records show how *thread* fields are used. The first of the two contains six *thread* fields that can be referenced by both *thread* and *fixup* fields in subsequent FIXUPP records in the same object module:

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  9C 0D 00 00 03 01 02 02 01 03 04 40 01 45 01 C0  ...........@.....
```

Bytes 03–04H, 05–06H, 07–08H, 09–0AH, 0B–0CH, and 0D–0EH represent the six *thread* fields in this FIXUPP record. The high-order bit of the first byte of each of these fields is 0, indicating that they are, indeed, *thread* fields and not *fixup* fields.

- Byte 03H, which contains 00H, is the *thread data* byte of the first *thread* field. Bit 7 of this byte is 0, indicating this is a *thread* field. Bit 6 (the *D* bit) is 0, so this field specifies a TARGET. Bit 5 is 0, as it must always be. Bits 4 through 2 (the *method* field) contain 0 (000B), which specifies TARGET method 0. Finally, bits 1 and 0 contain 0 (00B), the *thread number* that identifies this *thread* field.

  Byte 04H represents a segment *index* field, because method 0 of specifying a TARGET references a segment. The value of the index, 3, is a reference to the third SEGDEF record defined in the object module.
- Bytes 05–06H, 07–08H, and 09–0AH contain similar *thread* fields. In each, the *method* field specifies TARGET method 0. The three *thread* fields also have *thread numbers* of 1, 2, and 3. Because TARGET method 0 is specified for each *thread* field, bytes 06H, 08H, and 0AH represent segment *index* fields, which reference the second, first, and fourth SEGDEF records, respectively.
- Byte 0BH (the *thread data* byte of the fifth *thread* field in this FIXUPP record) contains 40H (01000000B). The *D* bit (bit 6) is 1, so this *thread* field specifies a FRAME. The *method* field (bits 4 through 2) contains 0 (000B), which specifies FRAME method 0. Byte 0CH (which contains 01H) is therefore interpreted as a segment *index* reference to the first SEGDEF record in the object module.
- Byte 0DH is the *thread data* byte of the sixth *thread* field. It contains 45H (01000101B). Bit 6 is 1, which indicates that this *thread* specifies a FRAME. The *method* field (bits 4 through 2) contains 1 (001B), which specifies FRAME method 1. Byte 0EH (which contains 01H) is therefore interpreted as a group *index* to the first preceding GRPDEF record.

  The *thread number* fields of the fifth and sixth *thread* fields contain 0 and 1, respectively, but these *thread numbers* do not conflict with the ones used in the first and second *thread* fields, because the latter represent TARGET references, not FRAME references.

The next FIXUPP example appears after the preceding record, in the same object module. This FIXUPP record contains a *fixup* field in bytes 03–05H that refers to a *thread* in the previous FIXUPP record:

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  9C 04 00 C4 09 9D F6                        .......
```

- Bytes 03–04H represent the 16-bit *locat* field, which contains C409H (1100010000001001B). Bit 15 of the *locat* field is 1, indicating a *fixup* field. The *M* bit (bit 14) is 1, so this fixup is relative to a particular segment, which is specified later in the *fixup* field. Bit 13 is 0, as it should be. Bits 12–10 (the *loc* field) contain 1 (001B), so the LOCATION to be fixed up is a 16-bit offset. Bits 9–0 (the *data record offset* field) contain 9 (0000001001B), so the LOCATION to be fixed up is represented at an offset of 9 bytes into the data field of the preceding LEDATA or LIDATA record.

- Byte 05H (the *fix dat* byte) contains 9DH (10011101B). The *F* bit (bit 7) is 1, so this *fixup* field references a *thread* field that, in turn, defines the method of specifying the FRAME for the fixup. Bits 6–4 (the *frame* field) contain 1 (001B), the number of the *thread* that contains the FRAME method. This *thread* contains a *method* number of 1, which references the first GRPDEF record in the object module, thus specifying the FRAME.

  The *T* bit (bit 3 in the *fix dat* byte) is 1, so the TARGET method is also defined in a preceding *thread* field. The *targt* field (bits 1 and 0 in the *fix dat* byte) contains 1 (01B), so the TARGET *thread* field whose *thread number* is 1 specifies the TARGET. The *P* bit (bit 3 in the *fix dat* byte) contains 1, which is combined with the low-order bits of the *method* field in the *thread* field that describes the target to obtain TARGET method number 4 (100B). The TARGET *thread* references the second SEGDEF record to specify the TARGET.

The last FIXUPP example illustrates that the linker performs a fixup by adding the calculated address value to the value in the LOCATION being fixed up. This function of the linker can be exploited to use fixups to modify opcodes or program data, as well as to resolve address references.

Consider how the following assembler instruction might be fixed up:

```
lea  bx,alpha+10h   ; alpha is an external symbol
```

Typically, this instruction is translated into an LEDATA record with zero in the LOCATION (bytes 08–09H) to be fixed up:

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  A0 08 00 01 00 00 8D 1E 00 00 AC                ...........
```

The corresponding FIXUPP record contains a *target displacement* of 10H bytes (bytes 08–09H):

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  9C 08 00 C4 02 02 01 01 10 00 82                ...........
```

This FIXUPP record specifies TARGET method 2, which is indicated by the *targt* field (bits 2–0) of the *fixdat* field (byte 05H). In this case, the linker adds the *target displacement* to the address it has determined for the TARGET (*alpha*) and then completes the fixup by adding this calculated address value to the zero value in the LOCATION.

The same result can be achieved by storing the displacement (10H) directly in the LOCATION in the LEDATA record:

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  A0 08 00 01 00 00 8D 1E 10 00 9C                ...........
```

Then, the *target displacement* can be omitted from the FIXUPP record:

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  9C 06 00 C4 02 06 01 01 90                      .........
```

This FIXUPP record specifies TARGET method 6, which does not use a *target displacement.* The linker performs this fixup by adding the address of *alpha* to the value in the LOCATION, so the result is identical to the preceding one.

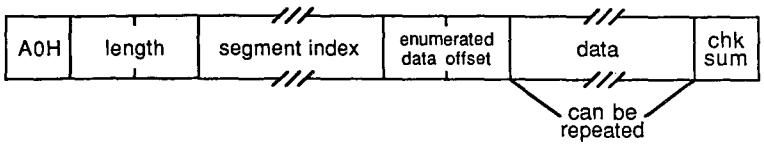
The difference between the two techniques is that in the latter the linker does not perform error checking when it adds the calculated fixup value to the value in the LOCATION. If this second technique is used, the linker will not flag arithmetic overflow or underflow errors when it adds the displacement to the TARGET address. The first technique, then, traps all errors; the second can be used when overflow or underflow is irrelevant and an error message would be undesirable.

0A0H LEDATA Logical Enumerated Data Record

The LEDATA record contains contiguous binary data — executable code or program data — that is eventually copied into the program's executable binary image.

The binary data in an LEDATA record can be modified by the linker if the record is followed by a FIXUPP record.

Record format



Segment index

The *segment index* is a variable-length index field. The index number in this field refers to a previous SEGDEF record in the object module. A value of 1 indicates the first SEGDEF record, a value of 2 the second, and so on. That SEGDEF record, in turn, indicates the segment into which the data in this LEDATA record is to be placed.

Enumerated data offset

The *enumerated data offset* is a 2-byte offset into the segment referenced by the *segment index*, relative to the base of the segment. Taken together, the *segment index* and the *enumerated data offset* fields indicate the location where the enumerated data will be placed in the run-time memory map.

Data

The *data* field contains the actual data, which can be either executable 8086 instructions or program data. The maximum size of the *data* field is 1024 bytes.

Location in object module

Any LEDATA records in an object module must be preceded by the SEGDEF records to which they refer. Also, if an LEDATA record requires a fixup, a FIXUPP record must immediately follow the LEDATA record.

Example

The following LEDATA record contains a simple text string:

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  A0 13 00 02 00 00 48 65 6C 6C 6F 2C 20 77 6F 72   ......Hello, wor
0010  6C 64 0D 0A 24 A8                                 ld..$.
```

- Byte 00H contains 0A0H, which identifies this as an LEDATA record.
- Bytes 01–02H contain 0013H, the length of the remainder of the record.

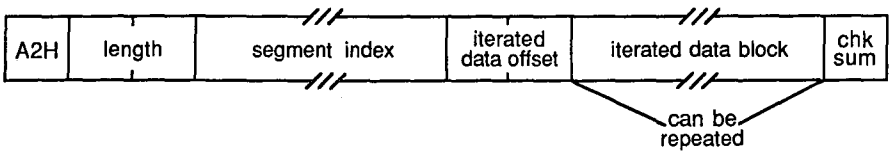
- Byte 03H (the *segment index* field) contains 02H, a reference to the second SEGDEF record in the object module.
- Bytes 04–05H (the *enumerated data offset* field) contain 0000H. This is the offset, from the base of the segment indicated by the *segment index* field, at which the data in the *data* field will be placed when the program is linked. Of course, this offset is subject to relocation by the linker because the segment declared in the specified SEGDEF record may be relocatable and may be combined with other segments declared in other object modules.
- Bytes 06–14H (the *data* field) contain the actual data.
- Byte 15H contains the checksum, 0A8H.

0A2H LIDATA Logical Iterated Data Record

Like the LEDATA record, the LIDATA record contains binary data — executable code or program data. The data in an LIDATA record, however, is specified as a repeating pattern (iterated), rather than by explicit enumeration.

The data in an LIDATA record may be modified by the linker if the LIDATA record is followed by a FIXUPP record.

Record format



Segment index

The *segment index* is a variable-length index field. The index number in this field refers to a previous SEGDEF record in the object module. A value of 1 indicates the first SEGDEF record, 2 indicates the second, and so on. That SEGDEF record, in turn, indicates the segment into which the data in this LIDATA record is to be placed when the program is executed.

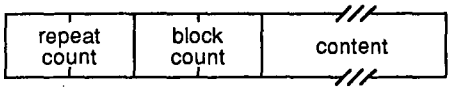
Iterated data offset

The *iterated data offset* is a 2-byte offset into the segment referenced by the *segment index*, relative to the base of the segment. Taken together, the *segment index* and the *iterated data offset* fields indicate the location where the iterated data will be placed in the run-time memory map.

Iterated data block

The *iterated data block* is a variable-length field containing the actual data — executable code and program data. *Iterated data blocks* can be nested, so one *iterated data block* can contain one or more other *iterated data blocks*. Microsoft LINK restricts the maximum size of an *iterated data block* to 512 bytes.

The format of the *iterated data block* is



- *Repeat count* is a 2-byte field indicating the number of times the *content* field is to be repeated.
- *Block count* is a 2-byte field indicating the number of *iterated data blocks* in the *content* field. If the *block count* is 0, the *content* field contains data only.

- *Content* is a variable-length field that can contain either nested *iterated data blocks* (if the *block count* is nonzero) or data (if the *block count* is 0). If the *content* field contains data, the field contains a 1-byte count of the number of data bytes in the field, followed by the actual data.

Location in object module

Any LIDATA records in an object module must be preceded by the SEGDEF records to which they refer. Also, if an LIDATA record requires a fixup, a FIXUPP record must immediately follow the LIDATA record.

Example

This sample LIDATA record corresponds to the following assembler statement, which declares a 10-element array containing the strings *ALPHA* and *BETA*:

```
db      10 dup('ALPHA','BETA')
```

The LIDATA record is

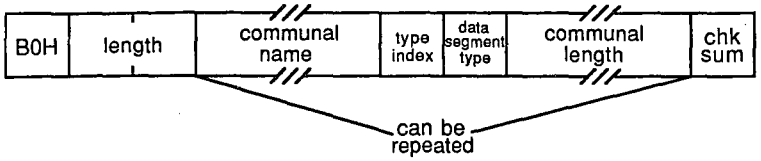
```
       0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  A2 1B 00 01 00 00 0A 00 02 00 01 00 00 00 05 41  ...............A
0010  4C 50 48 41 01 00 00 00 04 42 45 54 41 A9        LPHA.....BETA.
```

- Byte 00H contains 0A2H, identifying this as an LIDATA record.
- Bytes 01–02H contain 1BH, the length of the remainder of the record.
- Byte 03H (the *segment index*) contains 01H, a reference to the first SEGDEF record in this object module, indicating that the data declared in this LIDATA record is to be placed into the segment described by the first SEGDEF record.
- Bytes 04–05H (the *iterated data offset*) contain 0000H, so the data in this LIDATA record is to be located at offset 0000H in the segment designated by the *segment index*.
- Bytes 06–1CH represent an *iterated data block*:
  - Bytes 06–07H contain the *repeat count*, 000AH, which indicates that the *content* field of this *iterated data block* is to be repeated 10 times.
  - Bytes 08–09H (the *block count* for this *iterated data block*) contain 0002H, which indicates that the *content* field of this *iterated data block* (bytes 0A–1CH) contains two nested *iterated data block* fields (bytes 0A–13H and bytes 14–1CH).
  - Bytes 0A–0BH contain 0001H, the *repeat count* for the first nested *iterated data block*. Bytes 0C–0DH contain 0000H, indicating that the *content* field of this nested *iterated data block* contains data, rather than more nested *iterated data blocks*. The *content* field (bytes 0E–13H) contains the data: Byte 0EH contains 05H, the number of subsequent data bytes, and bytes 0F–13H contain the actual data (the string *ALPHA*).
  - Bytes 14–1CH represent the second nested *iterated data block*, which has a format similar to that of the block in bytes 0A–13H. This second nested *iterated data block* represents the 4-byte string *BETA*.
- Byte 1DH is the checksum, 0A9H.

0B0H COMDEF Communal Names Definition Record

The COMDEF record is a Microsoft extension to the basic set of 8086 object record types defined by Intel that declares a list of one or more communal variables. The COMDEF record is recognized by versions 3.50 and later of LINK. Microsoft encourages the use of the COMDEF record for declaration of communal variables.

Record format



Communal name

The *communal name* field is a variable-length field that contains the name of a communal variable. The first byte of this field indicates the length of the name contained in the remainder of the field.

Type index

The *type index* field is an index field that references a previous TYPDEF record in the object module. A value of 1 indicates the first TYPDEF record in the module, a value of 2 indicates the second, and so on. The *type index* value can be 0 if no data type is associated with the public name.

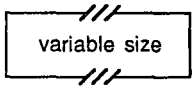
Data segment type

The *data segment type* field is a single byte that indicates whether the communal variable is FAR or NEAR. There are only two possible values for *data segment type*:

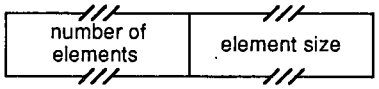
61H	FAR variable
62H	NEAR variable

Communal length

The *communal length* is a variable-length field that indicates the amount of memory to be allocated for the communal variable. The contents of this field depend on the value in the *data segment type* field. If the *data segment type* is NEAR (62H), the *communal length* field contains the size (in bytes) of the communal variable:



If the *data segment type* is FAR (61H), the *communal length* field is formatted as follows:

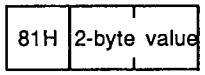


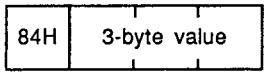
A FAR communal variable is viewed as an array of elements of a specified size. Thus, the *number of elements* field is a variable-length field representing the number of elements in the array, and the *element size* field is a variable-length field that indicates the size (in bytes) of each element. The amount of memory required for a FAR communal variable is thus the product of the *number of elements* and the *element size.*

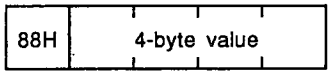
The format of the *variable size*, *number of elements*, and *element size* fields depends upon the magnitude of the values they contain:

- If the value is less than 128 (80H), the field is formatted as a 1-byte field containing the actual value:

- If the value is 128 (80H) or greater, the field is formatted with an extra initial byte that indicates whether the value is represented in the subsequent 2, 3, or 4 bytes:







Groups of *communal name*, *type index*, *data segment type*, and *communal length* fields can be repeated so that more than one communal variable can be declared in the same COMDEF record.

Location in object module

Any object module that contains COMDEF records must also contain one COMENT record with the *comment class* 0A1H, indicating that Microsoft extensions to the Intel object record specification are included in the object module. This COMENT record must appear before any COMDEF records in the object module.

Example

The following COMDEF record was generated by the Microsoft C Compiler version 4.0 for these public variable declarations:

```
int     foo;                    /* 2-byte integer */
char    foo2[32768];            /* 32768-byte array */
char    far foo3[10][2][20];    /* 400-byte array */
```

The COMDEF record is

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  B0 20 00 04 5F 66 6F 6F 00 62 02 05 5F 66 6F 6F   . ..._foo.b..._foo
0010  32 00 62 81 00 80 05 5F 66 6F 6F 33 00 61 81 90   2.b...._foo3.a..
0020  01 01 99                                         ...
```

- Byte 00H contains 0B0H, indicating that this is a COMDEF record.
- Bytes 01–02H contain 0020H, the length of the remainder of the record.
- Bytes 03–0AH, 0B–15H, and 16–21H represent three declarations for the communal variables *foo*, *foo2*, and *foo3*. The C compiler prepends an underscore to each of the names declared in the source code, so the symbols represented in this COMDEF record are *_foo*, *_foo2*, and *_foo3*.
  - Byte 03H contains 04H, the length of the first *communal name* in this record. Bytes 04–07H contain the name itself (*_foo*). Byte 08H (the *type index* field) contains 00H, as required. Byte 09H (the *data segment type* field) contains 62H, indicating this is a NEAR variable. Byte 0AH (the *communal length* field) contains 02H, the size of the variable in bytes.
  - Byte 0BH contains 05H, the length of the second *communal name*. Bytes 0C–10H contain the name, *_foo2*. Byte 11H is the *type index* field, which again contains 00H as required. Byte 12H (the *data segment type* field) contains 62H, indicating that *_foo2* is a NEAR variable.

    Bytes 13–15H (the *communal length* field) contain the size in bytes of the variable. The first byte of the *communal length* field (byte 13H) is 81H, indicating that the size is represented in the subsequent 2 bytes of data — bytes 14–15H, which contain the value 8000H.
  - Bytes 16–1BH represent the *communal name* field for *_foo3*, the third communal variable declared in this record. Byte 1CH (the *type index* field) again contains 00H as required. Byte 1DH (the *data segment type* field) contains 61H, indicating this is a FAR variable. This means the *communal length* field is formatted as a *number of elements* field (bytes 1E–20H, which contain the value 0190H) and an *element size* field (byte 21H, which contains 01H). The total size of this communal variable is thus 190H times 1, or 400 bytes.
- Byte 22H contains the checksum, 99H.

Richard Wilton

Article 20
The Microsoft Object Linker

MS-DOS object modules can be processed in two ways: They can be grouped together in object libraries, or they can be linked into executable files. All Microsoft language translators are distributed with two utility programs that process object modules: The Microsoft Library Manager (LIB) creates and modifies object libraries; the Microsoft Object Linker (LINK) processes the individual object records within object modules to create executable files.

The following discussion focuses on LINK because of its crucial role in creating an executable file. Before delving into the complexities of LINK, however, it is worthwhile reviewing how object modules are managed.

Object Files, Object Libraries, and LIB

Compilers and assemblers translate source-code modules into object modules (Figure 20-1). *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING TOOLS: Object Modules. An object module consists of a sequence of object records that describe the form and content of part of an executable program. An MS-DOS object module always starts with a THEADR record; subsequent object records in the module follow the sequence discussed in the Object Modules article.

Object modules can be stored in either of two types of MS-DOS files: object files and object libraries. By convention, object files have the filename extension .OBJ and object libraries have the extension .LIB. Although both object files and object libraries contain one or

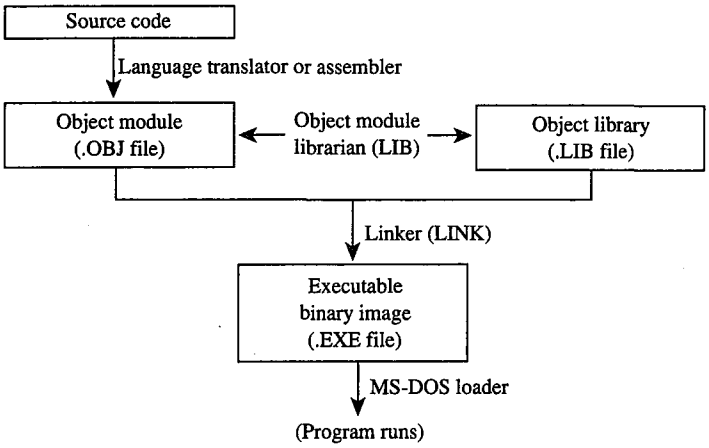


Figure 20-1. Object modules, object libraries, LIB, and LINK.

more object modules, the files and the libraries have different internal organization. Furthermore, LINK processes object files and libraries differently.

The structures of object files and libraries are compared in Figure 20-2. An object file is a simple concatenation of object modules in any arbitrary order. (Microsoft discourages the use of object files that contain more than one object module; Microsoft language translators never generate more than one object module in an object file.) In contrast, a library contains a hashed dictionary of all the public symbols declared in each of the object modules, in addition to the object modules themselves. Each symbol in the dictionary is associated with a reference to the object module in which the symbol was declared.

LINK processes object files differently than it does libraries. When LINK builds an executable file, it incorporates all the object modules in all the object files it processes. In contrast, when LINK processes libraries, it uses the hashed symbol dictionary in each library to extract object modules selectively — it uses an object module from a library only when the object module contains a symbol that is referenced within some other object module. This distinction between object files and libraries is important in understanding what LINK does.

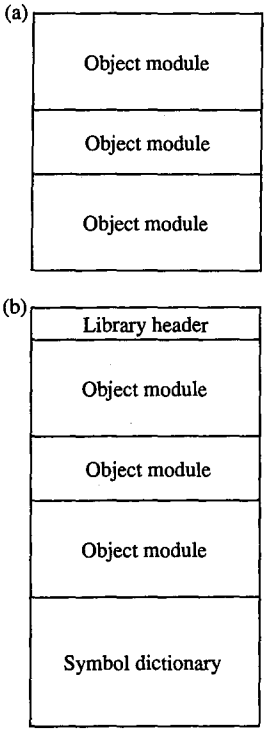


Figure 20-2. Structures of an object file and an object library. (a) An object file contains one or more object modules. (Microsoft discourages using more than one object module per object file.) (b) An object library contains one or more object modules plus a hashed symbol dictionary indicating the object modules in which each public symbol is defined.

What LINK Does

The function of LINK is to translate object modules into an executable program. LINK's input consists of one or more object files (.OBJ files) and, optionally, one or more libraries (.LIB files). LINK's output is an executable file (.EXE file) containing binary data that can be loaded directly from the file into memory and executed. LINK can also generate a symbolic address map listing (.MAP file)—a text file that describes the organization of the .EXE file and the correspondence of symbols declared in the object modules to addresses in the executable file.

Building an executable file

LINK builds two types of information into a .EXE file. First, it extracts executable code and data from the LEDATA and LIDATA records in object modules, arranges them in a specified order according to its rules for segment combination and relocation, and copies the result into the .EXE file. Second, LINK builds a header for the .EXE file. The header describes the size of the executable program and also contains a table of load-time segment relocations and initial values for certain CPU registers. *See* Pass 2 below.

Relocation and linking

In building an executable image from object modules, LINK performs two essential tasks: relocation and linking. As it combines and rearranges the executable code and data it extracts from the object modules it processes, LINK frequently adjusts, or relocates, address references to account for the rearrangements (Figure 20-3). LINK links object modules by resolving address references among them. It does this by matching the symbols declared in EXTDEF and PUBDEF object records (Figure 20-4). LINK uses FIXUPP records to determine exactly how to compute both address relocations and linked address references.

Object Module Order

LINK processes input files from three sources: object files and libraries specified explicitly by the user (in the command line, in response to LINK's prompts, or in a response file) and object libraries named in object module COMENT records.

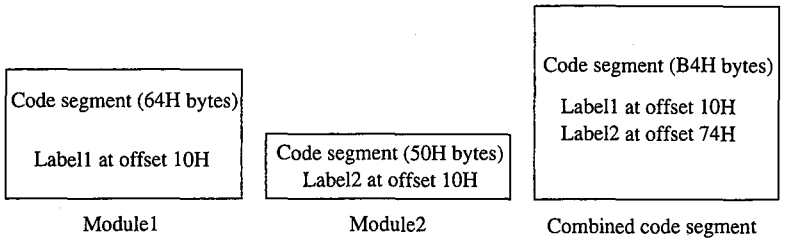


Figure 20-3. A simple relocation. Both object modules contain code that LINK combines into one logical segment. In this example, LINK appends the 50H bytes of code in Module2 *to the 64H bytes of code in* Module1. *LINK relocates all references to addresses in the code segment so that they apply to the combined segment.*

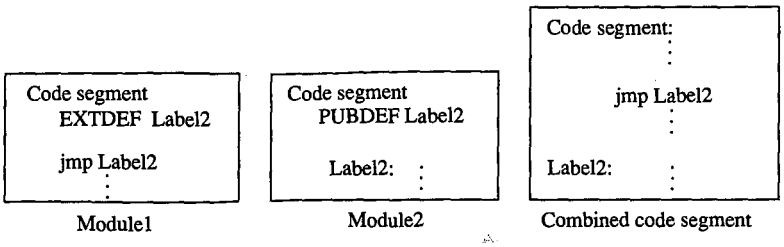


Figure 20-4. Resolving an external reference. LINK resolves the external reference in Module1 *(declared in an EXTDEF record) with the address of* Label2 *in* Module2 *(declared in a PUBDEF record).*

LINK always uses all the object modules in the object files it processes. In contrast, it extracts individual object modules from libraries—only those object modules needed to resolve references to public symbols are used. This difference is implicit in the order in which LINK reads its input files:

1. Object files specified in the command line or in response to the *Object Modules* prompt
2. Libraries specified in the command line or in response to the *Libraries* prompt
3. Libraries specified in COMENT records

The order in which LINK processes object modules influences the resulting executable file in three ways. First, the order in which segments appear in LINK's input files is reflected in the segment structure of the executable file. Second, the order in which LINK resolves external references to public symbols depends on the order in which it finds the public symbols in its input files. Finally, LINK derives the default name of the executable file from the name of the first input object file.

Segment order in the executable file

In general, LINK builds named segments into the executable file in the order in which it first encounters the SEGDEF records that declare the segments. (The /DOSSEG switch also affects segment order. *See* Using the /DOSSEG Switch below.) This means that the order in which segments appear in the executable file can be controlled by linking object modules in a specific order. In assembly-language programs, it is best to declare all the segments used in the program in the first object module to be linked so that the segment order in the executable file is under complete control.

Order in which references are resolved

LINK resolves external references in the order in which it encounters the corresponding public declarations. This fact is important because it determines the order in which LINK extracts object modules from libraries. When a public symbol required to resolve an external reference is declared more than once among the object modules in the input libraries, LINK uses the first object module that contains the public symbol. This means that the actual executable code or data associated with a particular external reference can be varied by changing the order in which LINK processes its input libraries.

For example, imagine that a C programmer has written two versions of a function named *myfunc()* that is called by the program MYPROG.C. One version of *myfunc()* is for debugging; its object module is found in MYFUNC.OBJ. The other is a production version whose object module resides in MYLIB.LIB. Under normal circumstances, the programmer links the production version of *myfunc()* by using MYLIB.LIB (Figure 20-5). To use the debugging version of *myfunc()*, the programmer explicitly includes its object module (MYFUNC.OBJ) when LINK is executed. This causes LINK to build the debugging version of *myfunc()* into the executable file because it encounters the debugging version in MYFUNC.OBJ before it finds the other version in MYLIB.LIB.

To exploit the order in which LINK resolves external references, it is important to know LINK's library search strategy: Each individual library is searched repeatedly (from first library to last, in the sequence in which they are input to LINK) until no further external references can be resolved.

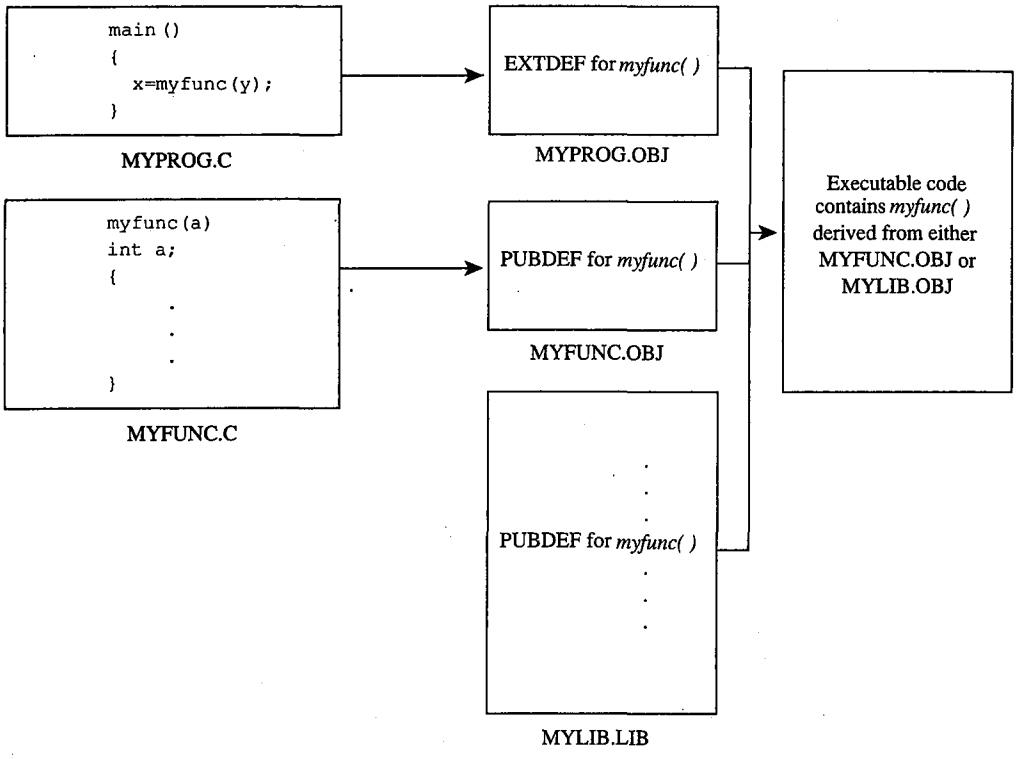


Figure 20-5. Ordered object module processing by LINK. (a) With the command LINK MYPROG,,,MYLIB, *the production version of* myfunc() *in MYLIB.LIB is used. (b) With the command* LINK MYPROG+ MYFUNC,,,MYLIB, *the debugging version of* myfunc() *in MYFUNC.OBJ is used.*

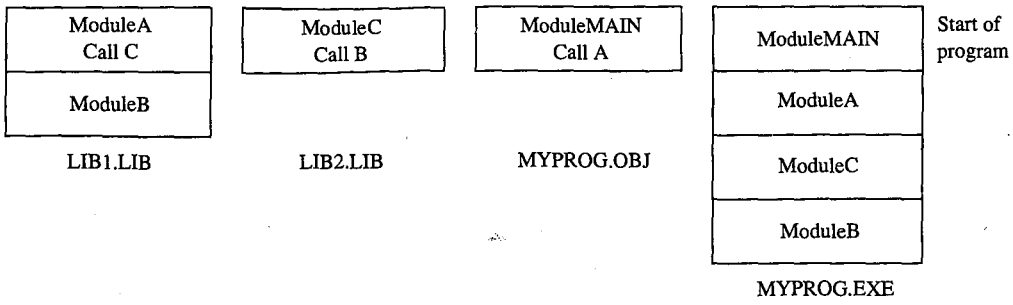


Figure 20-6. Library search order. Modules are incorporated into the executable file as LINK extracts them from the libraries to resolve external references.

The example in Figure 20-6 demonstrates this search strategy. Library LIB1.LIB contains object modules *A* and *B*, library LIB2.LIB contains object module *C*, and the object file MYPROG.OBJ contains the object module *MAIN*; modules *MAIN*, *A*, and *C* each contain an external reference to a symbol declared in another module. When this program is linked with

```
C>LINK MYPROG,,,LIB1+LIB2  <Enter>
```

LINK starts by incorporating the object module *MAIN* into the executable program. It then searches the input libraries until it resolves all the external references:

1. Process MYPROG.OBJ, find unresolved external reference to *A*.
2. Search LIB1.LIB, extract *A*, find unresolved external reference to *C*.
3. Search LIB1.LIB again; reference to *C* remains unresolved.
4. Search LIB2.LIB, extract *C*, find unresolved external reference to *B*.
5. Search LIB2.LIB again; reference to *B* remains unresolved.
6. Search LIB1.LIB again, extract *B*.
7. No more unresolved external references, so end library search.

The order in which the modules appear in the executable file thus reflects the order in which LINK resolves the external references; this, in turn, depends on which modules were contained in the libraries and on the order in which the libraries are input to LINK.

Name of the executable file

If no filename is specified in the command line or in response to the *Run File* prompt, LINK derives the name of the executable file from the name of the first object file it processes. For example, if the object files PROG1.OBJ and PROG2.OBJ are linked with the command

```
C>LINK PROG1+PROG2;  <Enter>
```

the resulting executable file, PROG1.EXE, takes its name from the first object file processed by LINK.

Segment Order and Segment Combinations

LINK builds segments into the executable file by applying the following sequence of rules:

1. Segments appear in the executable file in the order in which their SEGDEF declarations first appear in the input object modules.
2. Segments in different object modules are combined if they have the same name and class and a *public*, *memory*, *stack*, or *common* combine type. All address references within the combined segments are relocated relative to the start of the combined segment.
   - Segments with the same name and either the *public* or the *memory* combine type are combined in the order in which they are processed by LINK. The size of the resulting segment equals the total size of the combined segments.
   - Segments with the same name and the *stack* combine type are overlapped so that the data in each of the overlapped segments ends at the same address. The size of the resulting segment equals the total size of the combined segments. The resulting segment is always paragraph aligned.
   - Segments with the same name and the *common* combine type are overlapped so that the data in each of the overlapped segments starts at the same address. The size of the resulting segment equals the size of the largest of the overlapped segments.
3. Segments with the same class name are concatenated.
4. If the /DOSSEG switch is used, the segments are rearranged in conjunction with DGROUP. *See* Using the /DOSSEG Switch below.

These rules allow the programmer to control the organization of segments in the executable file by ordering SEGMENT declarations in an assembly-language source module, which produces the same order of SEGDEF records in the corresponding object module, and by placing this object module first in the order in which LINK processes its input files.

A typical MS-DOS program is constructed by declaring all executable code and data segments with the *public* combine type, thus enabling the programmer to compile the program's source code from separate source-code modules into separate object modules. When these object modules are linked, LINK combines the segments from the object modules according to the above rules to create logically unified code and data segments in the executable file.

Segment classes

LINK concatenates segments with the same class name after it combines segments with the same segment name and class. For example, Figure 20-7 shows the following compiling and linking:

```
C>MASM MYPROG1;  <Enter>
C>MASM MYPROG2;  <Enter>
C>LINK MYPROG1+MYPROG2;  <Enter>
```

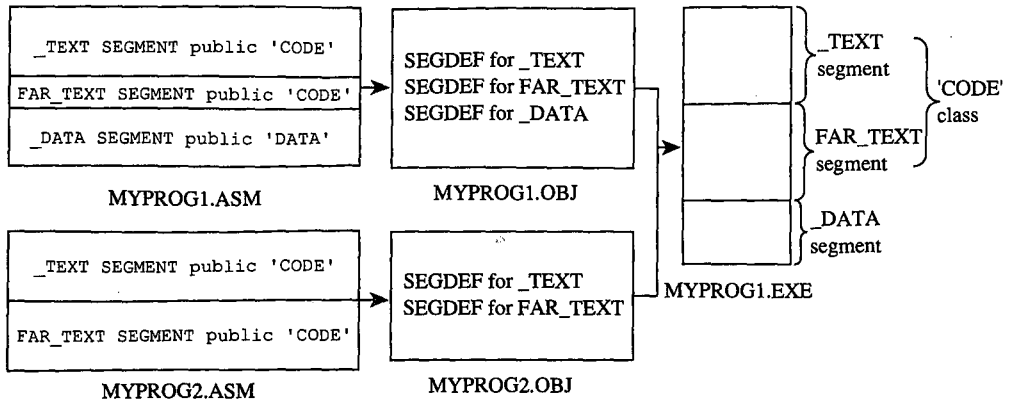


Figure 20-7. Segment order and concatenation by LINK. The start of each file, corresponding to the lowest address, is at the top.

After MYPROG1.ASM and MYPROG2.ASM have been compiled, LINK builds the *_TEXT* and *FAR_TEXT* segments by combining segments with the same name from the different object modules. Then, *_TEXT* and *FAR_TEXT* are concatenated because they have the same class name ('CODE'). *_TEXT* appears before *FAR_TEXT* in the executable file because LINK encounters the SEGDEF record for *_TEXT* before it finds the SEGDEF record for *FAR_TEXT*.

Segment alignment

LINK aligns the starting address of each segment it processes according to the alignment specified in each SEGDEF record. It adjusts the alignment of each segment it encounters regardless of how that segment is combined with other segments of the same name or class. (The one exception is *stack* segments, which always start on a paragraph boundary.)

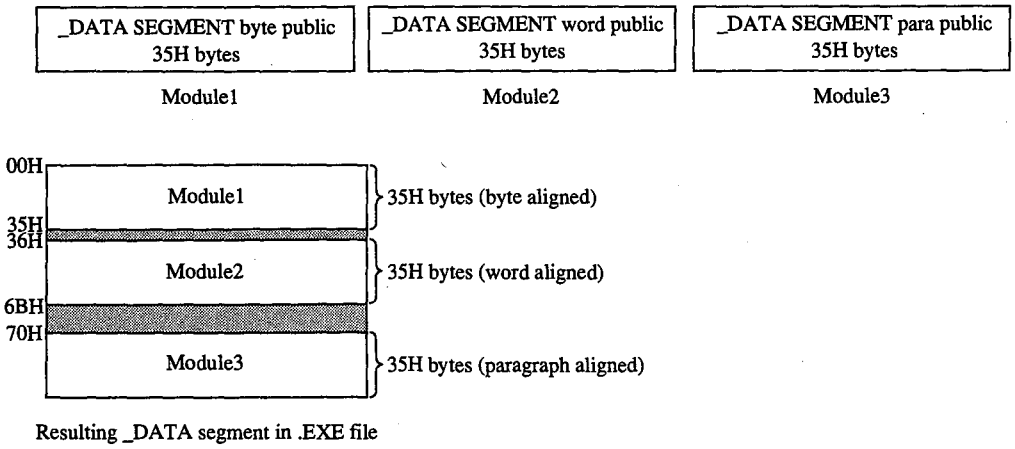


Figure 20-8. Alignment of combined segments. LINK enforces segment alignment by padding combined segments with uninitialized data bytes.

Segment alignment is particularly important when public segments with the same name and class are combined from different object modules. Note what happens in Figure 20-8, where the three concatenated *_DATA* segments have different alignments. To enforce the word alignment and paragraph alignment of the *_DATA* segments in *Module2* and *Module3*, LINK inserts one or more bytes of padding between the segments.

Segment groups

A segment group establishes a logical segment address to which all offsets in a group of segments can refer. That is, all addresses in all segments in the group can be expressed as offsets relative to the segment value associated with the group (Figure 20-9). Declaring segments in a group does not affect their positions in the executable file; the segments in a group may or may not be contiguous and can appear in any order as long as all address references to the group fall within 64 KB of each other.

```
DataGroup       GROUP     DataSeg1,DataSeg2
CodeSeg         SEGMENT   byte public 'CODE'
                ASSUME    cs:CodeSeg

                mov       ax,offset DataSeg2:TestData
                mov       ax,offset DataGroup:TestData

CodeSeg         ENDS

DataSeg1        SEGMENT   para public 'DATA'
                DB        100h dup(?)
DataSeg1        ENDS

DataSeg2        SEGMENT   para public 'DATA'
TestData        DB        ?
DataSeg2        ENDS
                END
```

Figure 20-9. Example of group addressing. The first MOV loads the value 00H into AX (the offset of TestData *relative to* DataSeg2*); the second MOV loads the value 100H into AX (the offset of* TestData *relative to the group* DataGroup*).*

LINK reserves one group name, DGROUP, for use by Microsoft language translators. DGROUP is used to group compiler-generated data segments and a default stack segment. *See* DGROUP below.

LINK Internals

Many programmers use LINK as a "black box" program that transforms object modules into executable files. Nevertheless, it is helpful to observe how LINK processes object records to accomplish this task.

LINK is a two-pass linker; that is, it reads all its input object modules twice. On Pass 1, LINK builds an address map of the segments and symbols in the object modules. On Pass 2, it extracts the executable code and program data from the object modules and builds a memory image—an exact replica—of the executable file.

The reason LINK builds an image of the executable file in memory, instead of simply copying code and data from object modules into the executable file, is that it organizes the executable file by segments and not by the order in which it processes object modules. The most efficient way to concatenate, combine, and relocate the code and data is to build a map of the executable file in memory during Pass 1 and then fill in the map with code and data during Pass 2.

In versions 3.52 and later, whenever the /I (/INFORMATION) switch is specified in the command line, LINK displays status messages at the start of each pass and as it processes each object module. If the /M (/MAP) switch is used in addition to the /I switch, LINK also displays the total length of each segment declared in the object modules. This information is helpful in determining how the structure of an executable file corresponds to the contents of the object modules processed by LINK.

Pass 1

During Pass 1, LINK processes the LNAMES, SEGDEF, GRPDEF, COMDEF, EXTDEF, and PUBDEF records in each input object module and uses the information in these object records to construct a symbol table and an address map of segments and segment groups.

Symbol table

As each object module is processed, LINK uses the symbol table to resolve external references (declared in EXTDEF and COMDEF records) to public symbols. If LINK processes all the object files without resolving all the external references in the symbol table, it searches the input libraries for public symbols that match the unresolved external references. LINK continues to search each library until all the external references in the symbol table are resolved.

Segments and groups

LINK processes each SEGDEF record according to the segment name, class name, and attributes specified in the record. LINK constructs a table of named segments and updates it as it concatenates or combines segments. This allows LINK to associate each public symbol in the symbol table with an offset into the segment in which the symbol is declared.

LINK also generates default segments into which it places communal variables declared in COMDEF records. Near communal variables are placed in one paragraph-aligned public segment named c_common, with class name BSS (block storage space) and group

DGROUP. Far communal variables are placed in a paragraph-aligned segment named FAR_BSS, with class name FAR_BSS. The combine type of each far communal variable's FAR_BSS segment is private (that is, not *public*, *memory*, *common*, or *stack*). As many FAR_BSS segments as necessary are generated.

After all the object files have been read and all the external references in the symbol table have been resolved, LINK has a complete map of the addresses of all segments and symbols in the program. If a .MAP file has been requested, LINK creates the file and writes the address map to it. Then LINK initiates Pass 2.

Pass 2

In Pass 2, LINK extracts executable code and program data from the LEDATA and LIDATA records in the object modules. It builds the code and data into a memory image of the executable file. During Pass 2, LINK also carries out all the address relocations and fixups related to segment relocation, segment grouping, and resolution of external references, as well as any other address fixups specified explicitly in object module FIXUPP records.

If it determines during Pass 2 that not enough RAM is available to contain the entire image, LINK creates a temporary file in the current directory on the default disk drive. (LINK versions 3.60 and later use the environment variable TMP to find the directory for the temporary scratch file.) LINK then uses this file in addition to all the available RAM to construct the image of the executable file. (In versions of MS-DOS earlier than 3.0, the temporary file is named VM.TMP; in versions 3.0 and later, LINK uses Interrupt 21H Function 5AH to create the file.)

LINK reads each of the input object modules in the same order as it did in Pass 1. This time it copies the information from each object module's LEDATA and LIDATA records into the memory image of each segment in the proper sequence. This is when LINK expands the iterated data in each LIDATA record it processes.

LINK processes each LEDATA and LIDATA record along with the corresponding FIXUPP record, if one exists. LINK processes the FIXUPP record, performs the address calculations required for relocation, segment grouping, and resolving external references, and then stores binary data from the LEDATA or LIDATA record, including the results of the address calculations, in the proper segment in the memory image. The only exception to this process occurs when a FIXUPP record refers to a segment address. In this case, LINK adds the address of the fixup to a table of segment fixups; this table is used later to generate the segment relocation table in the .EXE header.

When all the data has been extracted from the object modules and all the fixups have been carried out, the memory image is complete. LINK now has all the information it needs to build the .EXE header (Table 20-1). At this point, therefore, LINK creates the executable file and writes the header and all segments into it.

Table 20-1. How LINK Builds a .EXE File Header.

Offset	Contents	Comments
00H	'MZ'	.EXE file signature
02H	Length of executable image MOD 512	Total size of all segments plus .EXE file header
04H	Length of executable image in 512-byte pages, including last partial page (if any)	
06H	Number of run-time segment relocations	Number of segment fixups
08H	Size of the .EXE header in 16-byte paragraphs	Size of segment relocation table
0AH	MINALLOC: Minimum amount of RAM to be allocated above end of the loaded program (in 16-byte paragraphs)	Size of uninitialized data and/or stack segments at end of program (0 if /HI switch is used)
0CH	MAXALLOC: Maximum amount of RAM to be allocated above end of the loaded program (in 16-byte paragraphs)	0 if /HI switch is used; value specified with /CP switch; FFFFH if /CP and /HI switches are not used
0EH	Stack segment (initial value for SS register); relocated by MS-DOS when program is loaded	Address of stack segment relative to start of executable image
10H	Stack pointer (initial value for register SP)	Size of stack segment in bytes
12H	Checksum	One's complement of sum of all words in file, excluding checksum itself
14H	Entry point offset (initial value for register IP)	MODEND object record that specifies program start address
16H	Entry point segment (initial value for register CS); relocated by MS-DOS when program is loaded	
18H	Offset of start of segment relocation table relative to start of .EXE header	
1AH	Overlay number	0 for resident segments; >0 for overlay segments
1CH	Reserved	

Using LINK to Organize Memory

By using LINK to rearrange and combine segments, a programmer can generate an executable file in which segment order and addressing serve specific purposes. As the following examples demonstrate, careful use of LINK leads to more efficient use of memory and simpler, more efficient programs.

Segment order for a TSR

In a terminate-and-stay-resident (TSR) program, LINK must be used carefully to generate segments in the executable file in the proper order. A typical TSR program consists of a resident portion, in which the TSR application is implemented, and a transient portion, which executes only once to initialize the resident portion. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: CUSTOMIZING MS-DOS: Terminate-and-Stay-Resident Utilities.

Because the transient portion of the TSR program is executed only once, the memory it occupies should be freed after the resident portion has been initialized. To allow the MS-DOS Terminate and Stay Resident function (Interrupt 21H Function 31H) to free this memory when it leaves the resident portion of the TSR program in memory, the TSR program must have its resident portion at lower addresses than its transient portion.

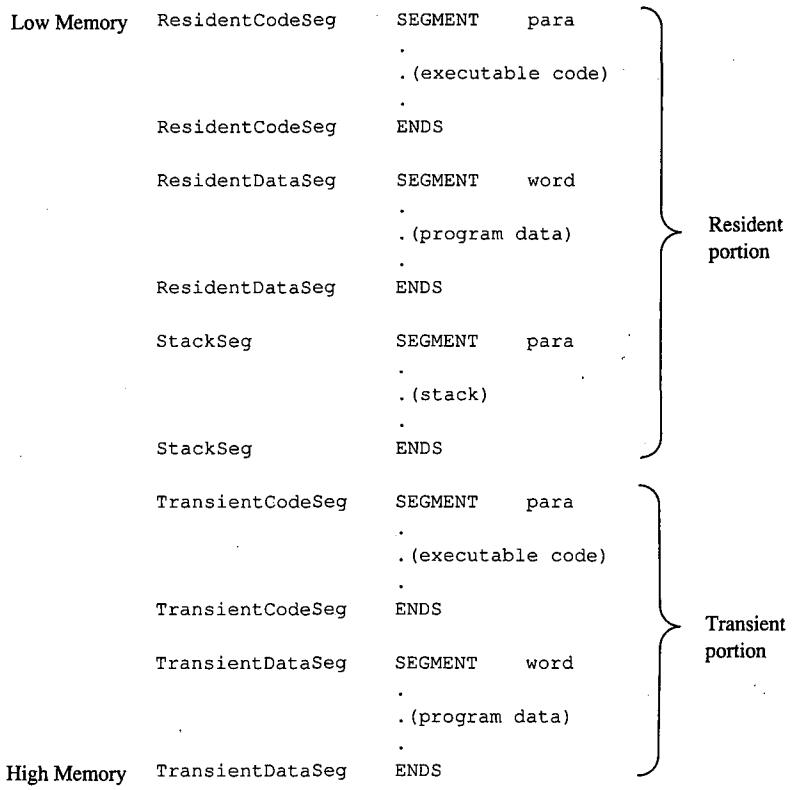


Figure 20-10. Segment order for a terminate-and-stay-resident program.

In Figure 20-10, the segments containing the resident code and data are declared before the segments that represent the transient portion of the program. Because LINK preserves this segment order, the executable program has the desired structure, with resident code and data at lower addresses than transient code and data. Moreover, the number of paragraphs in the resident portion of the program, which must be computed before Interrupt 21H Function 31H is called, is easy to derive from the segment structure: This value is the difference between the segment address of the program segment prefix, which immediately precedes the first segment in the resident portion, and the address of the first segment in the transient portion of the program.

Groups for unified segment addressing

In some programs it is desirable to maintain executable code and data in separate logical segments but to address both code and data with the same segment register. For example, in a hardware interrupt handler, using the CS register to address program data is generally simpler than using DS or ES.

In the routine in Figure 20-11, code and data are maintained in separate segments for program clarity, yet both can be addressed using the CS register because both code and data segments are included in the same group. (The SNAP.ASM listing in the Terminate-and-Stay-Resident Utilities article is another example of this use of a group to unify segment addressing.)

```
ISRgroup        GROUP     CodeSeg,DataSeg
CodeSeg         SEGMENT   byte public 'CODE'
                ASSUME    cs:ISRgroup
                mov       ax,offset ISRgroup:CodeLabel
CodeLabel:      mov       bx,ISRgroup:DataLabel
CodeSeg         ENDS

DataSeg         SEGMENT   para public 'DATA'
DataLabel       DW        ?
DataSeg         ENDS
                END
```

Figure 20-11. Code and data included in the same group. In this example, addresses within both CodeSeg *and* DataSeg *are referenced relative to the CS register by grouping the segments (using the assembler GROUP directive) and addressing the group through CS (using the assembler ASSUME directive).*

Uninitialized data segments

A segment that contains only uninitialized data can be processed by LINK in two ways, depending on the position of the segment in the program. If the segment is not at the end of the program, LINK generates a block of bytes initialized to zero to represent the segment in the executable file. If the segment appears at the end of the program, however, LINK does not generate a block of zeroed bytes. Instead, it increases the minimum run-time memory allocation by increasing MINALLOC (specified at offset 0AH in the .EXE header) by the amount of memory required for the segment.

Therefore, if it is necessary to reserve a large amount of uninitialized memory in a segment, the size of the .EXE file can be decreased by building the segment at the end of a program (Figure 20-12). This is why, for example, Microsoft high-level-language translators always build BSS and STACK segments at the end of compiled programs. (The loader does not fill these segments with zeros; a program must still initialize them with appropriate values.)

```
(a)             CodeSeg     SEGMENT     byte public 'CODE'
                            ASSUME      cs:CodeSeg,ds:DataSeg
                            ret
                CodeSeg     ENDS

                DataSeg     SEGMENT     word public 'DATA'
                BigBuffer   DB          10000 dup(?)
                DataSeg     ENDS
                            END

(b)             DataSeg     SEGMENT     word public 'DATA'
                BigBuffer   DB          10000 dup(?)
                DataSeg     ENDS

                CodeSeg     SEGMENT     byte public 'CODE'
                            ASSUME      cs:CodeSeg,ds:DataSeg
                            ret
                CodeSeg     ENDS
                            END
```

Figure 20-12. LINK processing of uninitialized data segments. (a) When DataSeg, *which contains only uninitialized data, is placed at the end of this program, the size of the .EXE file is only 513 bytes. (b) When* DataSeg *is not placed at the end of the program, the size of the .EXE file is 10513 bytes.*

Overlays

If a program contains two or more subroutines that are mutually independent—that is, subroutines that do not transfer control to each other—LINK can be instructed to build each subroutine into a separately loaded portion of the executable file. (This instruction is indicated in the command line when LINK is executed by enclosing each overlay subroutine or group of subroutines in parentheses.) Each of the subroutines can then be overlaid as it is needed in the same area of memory (Figure 20-13). The amount of memory required to run a program that uses overlays is, therefore, less than the amount required to run the same program without overlays.

A program that uses overlays must include the Microsoft run-time overlay manager. The overlay manager is responsible for copying overlay code from the executable file into memory whenever the program attempts to transfer control to code in an overlay. A program that uses overlays runs slower than a program that does not use them, because it takes longer to extract overlays separately from the .EXE file than it does to read the entire .EXE file into memory at once.

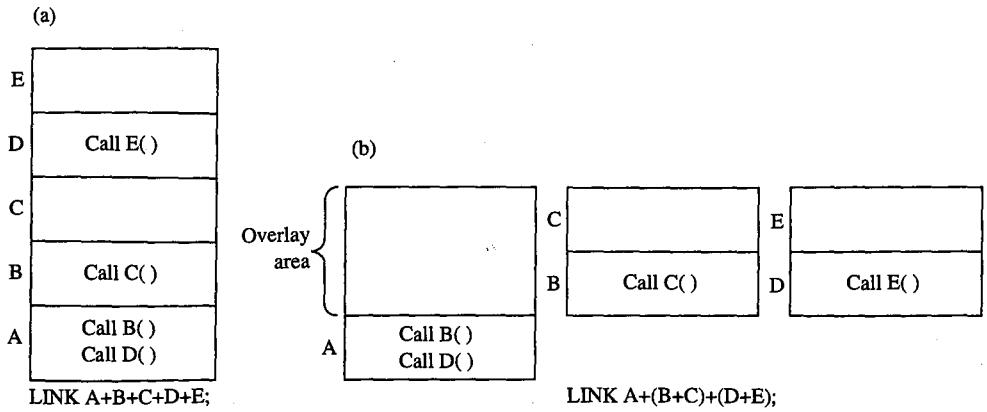


Figure 20-13. Memory use in a program linked (a) without overlays and (b) with overlays. In (b), either modules (B+C) or modules (D+E) can be loaded into the overlay area at run time.

The default object libraries that accompany Microsoft high-level-language compilers contain object modules that support the Microsoft run-time overlay manager. The following description of LINK's relationship to the run-time overlay manager applies to versions 3.00 through 3.60 of LINK; implementation details may vary in future versions.

Overlay format in a .EXE file

An executable file that contains overlays has a .EXE header preceding each overlay (Figure 20-14). The overlays are numbered in sequence, starting at 0; the overlay number is stored in the word at offset 1AH in each overlay's .EXE header. When the contents of the .EXE file are loaded into memory for execution, only the resident, nonoverlaid part of the program is copied into memory. The overlays must be read into memory from the .EXE file by the run-time overlay manager.

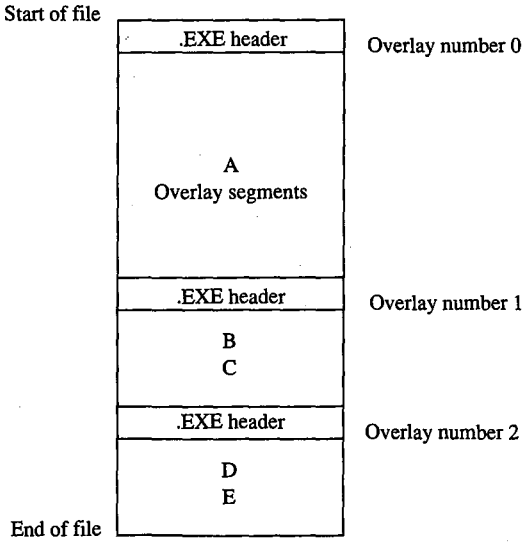


Figure 20-14. .EXE file structure produced by LINK A + (B+C) + (D+E).

Segments for overlays

When LINK produces an executable file that contains overlays, it adds three segments to those defined in the object modules: OVERLAY_AREA, OVERLAY_END, and OVERLAY_DATA. LINK assigns the segment class name 'CODE' to OVERLAY_AREA and OVERLAY_END and includes OVERLAY_DATA in the default group DGROUP.

OVERLAY_AREA is a reserved segment into which the run-time overlay manager is expected to load each overlay as it is needed. Therefore, LINK sets the size of OVERLAY_AREA to fit the largest overlay in the program. The OVERLAY_END segment is declared immediately after OVERLAY_AREA, so a program can determine the size of the OVERLAY_AREA segment by subtracting its segment address from that of OVERLAY_END. The OVERLAY_DATA segment is initialized by LINK with information about the executable file, the number of overlays, and other data useful to the run-time overlay manager.

LINK requires the executable code used in overlays to be contained in segments whose class names end in *CODE* and whose segment names differ from those of the segments used in the resident (nonoverlaid) portion of the program. In assembly language, this is accomplished by using the SEGMENT directive; in high-level languages, the technique of ensuring unique segment names depends on the compiler. In Microsoft C, for example, the /A switch in the command line selects the memory model and thus the segment naming defaults used by the compiler; in medium, large, and huge memory models, the compiler generates a unique segment name for each C function in the source code. In Microsoft FORTRAN, on the other hand, the compiler always generates a uniquely named segment for each SUBROUTINE and FUNCTION in the source code, so no special programming is required.

LINK substitutes all far CALL instructions from root to overlay or from overlay to overlay with a software interrupt followed by an overlay number and an offset into the overlay segment (Figure 20-15). The interrupt number can be specified with LINK's /OVERLAYINTERRUPT switch; if the switch is omitted, LINK uses Interrupt 3FH by default. By replacing calls to overlay code with a software interrupt, LINK provides a mechanism for the run-time overlay manager to take control, load a specified overlay into memory, and transfer control to a specified offset within the overlay.

```
(a)         EXTRN     OverlayEntryPoint:far
            call      OverlayEntryPoint    ; far CALL

(b)         int       IntNo                ; interrupt number
                                           ;  specified with /OVERLAYINTERRUPT
                                           ;  switch (default 3FH)
            DB        OverlayNumber        ; overlay number
            DW        OverlayEntry         ; offset of overlay entry point
                                           ;  (the address to which
                                           ;  the overlay manager transfers
                                           ;  control)
```

Figure 20-15. Executable code modification by LINK for accessing overlays. (a) Code as written. (b) Code as modified by LINK.

Run-time processing of overlays

The resident (nonoverlaid) portion of a program that uses overlays initializes the overlay interrupt vector specified by LINK with the address of the run-time overlay manager. (The OVERLAY_DATA segment contains the interrupt number.) The overlay manager then takes control wherever LINK has substituted a software interrupt for a far call in the executable code.

Each time the overlay manager executes, its first task is to determine which overlay is being called. It does this by using the return address left on the stack by the INT instruction that invoked the overlay manager; this address points to the overlay number stored in the byte after the interrupt instruction that just executed. The overlay manager then determines whether the destination overlay is already resident and loads it only if necessary. Next, the overlay manager opens the .EXE file, using the filename in the OVERLAY_DATA segment. It locates the start of the specified overlay in the file by examining the length (offset 02H and offset 04H) and overlay number (offset 1AH) in each overlay's .EXE header.

The overlay manager can then read the overlay from the .EXE file into the OVERLAY_AREA segment. It uses the overlay's segment relocation table to fix up any segment references in the overlay. The overlay manager transfers control to the overlay with a far call to the OVERLAY_AREA segment, using the offset stored by LINK 1 byte after the interrupt instruction (*see* Figure 20-15).

Interrupt 21H Function 4BH

LINK's protocol for implementing overlays is not recognized by Interrupt 21H Function 4BH (Load and Execute Program). This MS-DOS function, when called with AL = 03H, loads an overlay from a .EXE file into a specified location in memory. *See* SYSTEM CALLS: INTERRUPT 21H: Function 4BH. However, Function 4BH does not use an overlay number, so it cannot find overlays in a .EXE file formatted by LINK with multiple .EXE headers.

DGROUP

LINK always includes DGROUP in its internal table of segment groups. In object modules generated by Microsoft high-level-language translators, DGROUP contains both the default data segment and the stack segment. LINK's /DOSSEG and /DSALLOCATE switches both affect the way LINK treats DGROUP. Changing the way LINK manages DGROUP ultimately affects segment order and addressing in the executable file.

Using the /DOSSEG switch

The /DOSSEG switch causes LINK to arrange segments in the default order used by Microsoft high-level-language translators:

1. All segments with a class name ending in *CODE*. These segments contain executable code.
2. All other segments outside DGROUP. These segments typically contain far data items.

3. DGROUP segments. These are a program's near data and stack segments. The order in which segments appear in DGROUP is
   - Any segments of class BEGDATA. (This class name is reserved for Microsoft use.)
   - Any segments not of class BEGDATA, BSS, or STACK.
   - Segments of class BSS.
   - Segments of class STACK.

This segment order is necessary if programs compiled by Microsoft translators are to run properly. The /DOSSEG switch can be used whenever an object module produced by an assembler is linked ahead of object modules generated by a Microsoft compiler, to ensure that segments in the executable file are ordered as in the preceding list regardless of the order of segments in the assembled object module.

When the /DOSSEG switch is in effect, LINK always places DGROUP at the end of the executable program, with all uninitalized data segments at the end of the group. As discussed above, this placement helps to minimize the size of the executable file. The /DOSSEG switch also causes LINK to restructure the executable program to support certain conventions used by Microsoft language translators:

- Compiler-generated segments with the class name BEGDATA are placed at the beginning of DGROUP.
- The public symbols _edata and _end are generated to point to the beginning of the BSS and STACK segments.
- Sixteen bytes of zero are inserted in front of the _TEXT segment.

Microsoft compilers that rely on /DOSSEG conventions generate a special COMENT object record that sets the /DOSSEG switch when the record is processed by LINK.

Using the /HIGH and /DSALLOCATE switches

When a program has been linked without using LINK's /HIGH switch, MS-DOS loads program code and data segments from the .EXE file at the lowest address in the first available block of RAM large enough to contain the program (Figure 20-16). The value in the .EXE header at offset 0CH specifies the maximum amount of extra RAM MS-DOS must allocate to the program above what is loaded from the .EXE file. Above that, all unused RAM is managed by MS-DOS. With this memory allocation strategy, a program can use Interrupt 21H Functions 48H (Allocate Memory Block) and 4AH (Resize Memory Block) to increase or decrease the amount of RAM allocated to it.

When a program is linked with LINK's /HIGH switch, LINK zeros the words it stores in the .EXE header at offset 0AH and 0CH. Setting the words at 0AH and 0CH to zero indicates that the program is to be loaded into RAM at the highest address possible (Figure 20-16). With this memory layout, however, a program can no longer change its memory allocation dynamically because all available RAM is allocated to the program when it is loaded and the uninitialized RAM between the program segment prefix and the program itself cannot be freed.

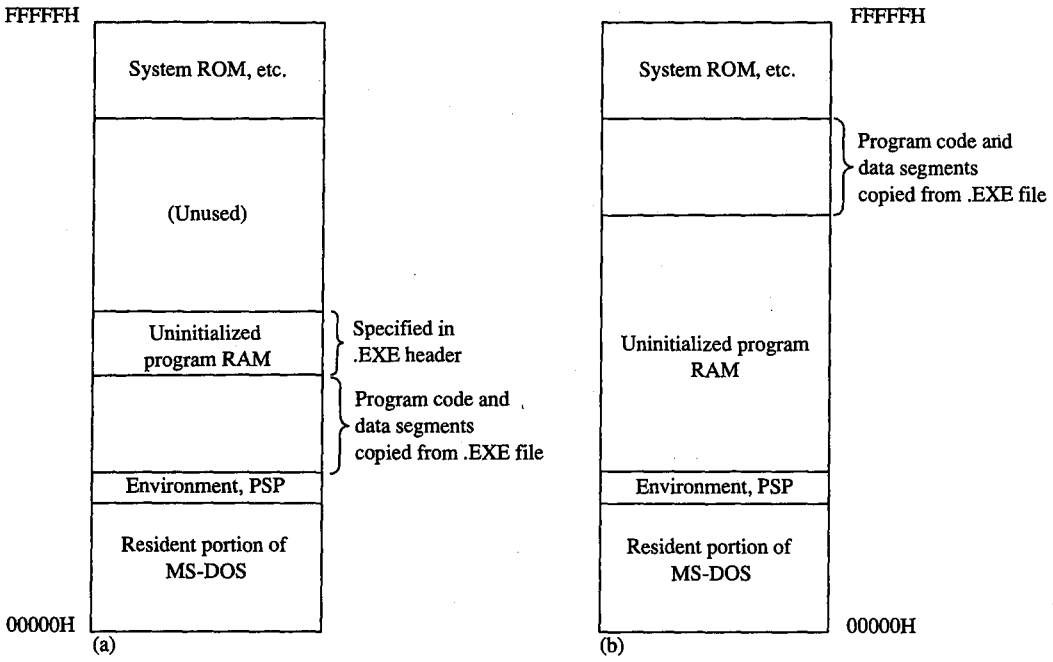


Figure 20-16. Effect of the /HIGH switch on run-time memory use. (a) The program is linked without *the /HIGH switch. (b) The program is linked* with *the /HIGH switch.*

The only reason to load a program with this type of memory allocation is to allow a program data structure to be dynamically extended toward lower memory addresses. For example, both stacks and heaps can be implemented in this way. If a program's stack segment is the first segment in its memory map, the stack can grow downward without colliding with other program data.

To facilitate addressing in such a segment, LINK provides the /DSALLOCATE switch. When a program is linked using this switch, all addresses within DGROUP are relocated in such a way that the last byte in the group has offset FFFFH. For example, if the program in Figure 20-17 is linked without the /DSALLOCATE and /HIGH switches, the value of offset *DGROUP:DataItem* would be 00H; if these switches are used, the linker adjusts the segment value of DGROUP downward so that the offset of *DataItem* within DGROUP becomes FFF0H.

Early versions of Microsoft Pascal (before version 3.30) and Microsoft FORTRAN (before version 3.30) generated object code that had to be linked with the /DSALLOCATE switch. For this reason, LINK sets the /DSALLOCATE switch by default if it processes an object module containing a COMENT record generated by one of these compilers. (Such a COMENT record contains the string *MS PASCAL* or *FORTRAN 77. See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING TOOLS: Object Modules.) Apart from this special requirement of certain language translators, however, the use of /DSALLOCATE and /HIGH should probably be avoided because of the limitations they place on run-time memory allocation.

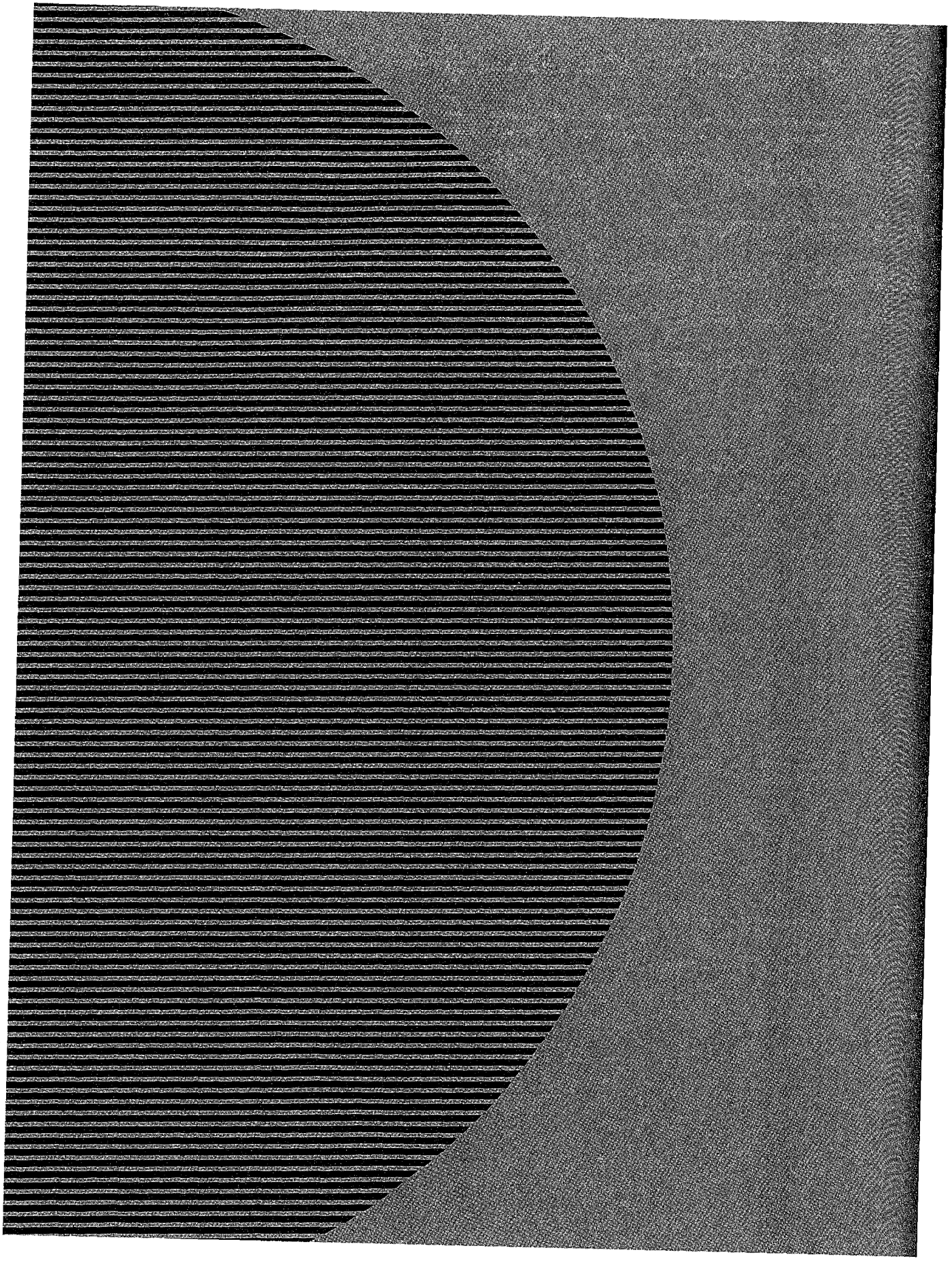
```
DGROUP          GROUP     _DATA
_DATA           SEGMENT   word public 'DATA'
DataItem        DB        10h dup (?)
_DATA           ENDS

_TEXT           SEGMENT   byte public 'CODE'
                ASSUME    cs:_TEXT,ds:DGROUP
                mov       bx,offset DGROUP:DataItem
_TEXT           ENDS
                END
```

Figure 20-17. The value of offset DGROUP:DataItem *in this program is FFF0H if the program is linked with the /DSALLOCATE switch or 00H if the program is linked without using the switch.*

Summary

LINK's characteristic support for segment ordering, for run-time memory management, and for dynamic overlays has an impact in many different situations. Programmers who write their own language translators must bear in mind the special conventions followed by LINK in support of Microsoft language translators. Application programmers must be familiar with LINK's capabilities when they use assembly language or link assembly-language programs with object modules generated by Microsoft compilers. LINK is a powerful program development tool and understanding its special capabilities can lead to more efficient programs.

Richard Wilton



Section III
User Commands

Introduction

This section of *The MS-DOS Encyclopedia* describes the standard internal and external MS-DOS commands available to the user who is running MS-DOS (versions 1.0 through 3.2). System configuration options, special batch-file directives, the line editor (EDLIN), and the installable device drivers normally included with MS-DOS are also covered.

Entries are arranged alphabetically by the name of the command or driver. The configuration, batch-file, and line-editor directives appear alphabetically under the headings CONFIG.SYS, BATCH, and EDLIN, respectively. Each entry includes

- Command name
- Version dependencies and network information
- Command purpose
- Prototype command and summary of options
- Detailed description of command
- One or more examples of command use
- Return codes (where applicable)
- Informational and error messages

The experienced user can find information with a quick glance at the first part of a command entry; a less experienced user can refer to the detailed explanation and examples in a more leisurely fashion. The next two pages contain an example of a typical entry from the User Commands section, with explanations of each component. This example is followed by listings of the commands by functional group.

The following terms are used for command-line variables in the sample syntax:

drive	a letter in the range A–Z, followed by a colon, indicating a logical disk drive.
path	a specific location in a disk's hierarchical directory structure; can include the special directory names . and ..; elements are separated by backslash characters (\).
pathname	a file specification that can include a path and/or drive and/or filename extension.
filename	the name of a file, generally with its extension; cannot include a drive or path.

Note: PC-DOS, though not an official product name, is used in this section to indicate IBM's version of the disk operating system originally provided by Microsoft. Commands sometimes have slightly different options or appear for the first time in different versions of MS-DOS and PC-DOS. When a command appears only in the IBM versions, the abbreviation IBM appears in the heading area. Significant differences between MS-DOS and PC-DOS versions of a command are indicated in the *Syntax* and *Description* portions of the entry.

HEADING
The command name as the user would enter it or as it would be used in a batch or system-configuration file.

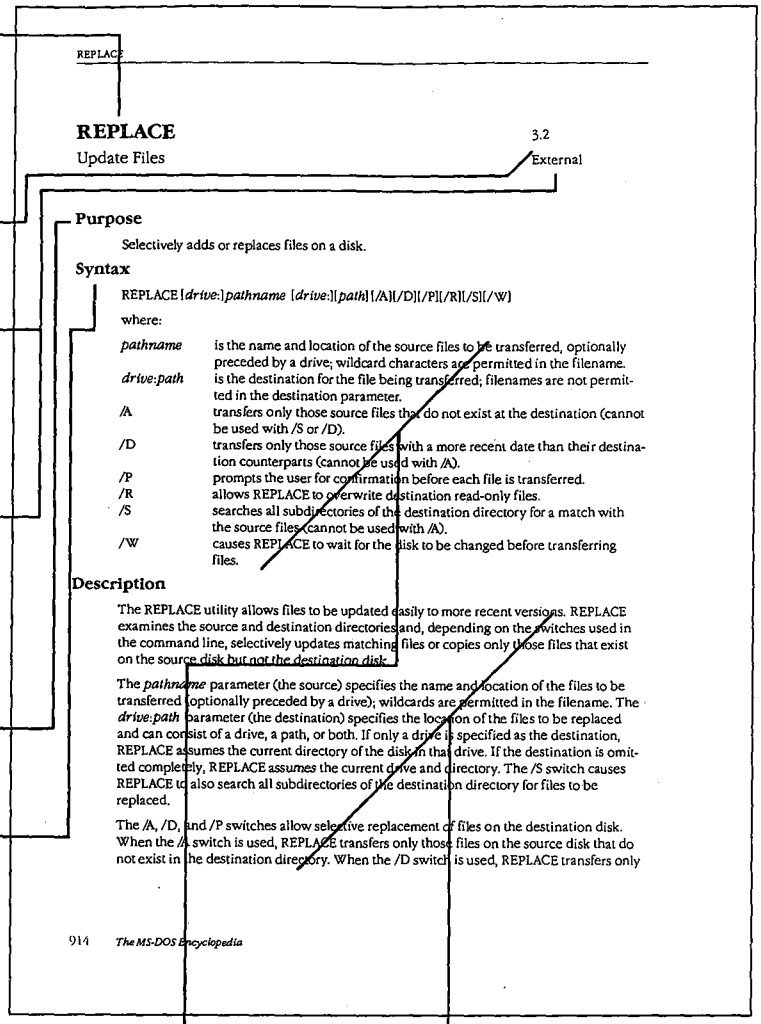
ICON-1
MS-DOS version dependency.

ICON-2
Whether the command is internal (built into COMMAND.COM) or external (loaded from a disk file when needed).

ICON-3
The abbreviation IBM if the command is present only in PC-DOS and the warning *No Net* if the command cannot be used across a network.

PURPOSE
An abstract of command purpose and usage.

SYNTAX
A prototype command line, with variable names in italic and optional parameters in square brackets. The various elements of the command line should be entered in the order shown. Any punctuation must be used exactly as shown; in commands that use commas as separators, the comma usually must be included as a placeholder even if the parameter is omitted. Except where noted, commands, parameters, and switches can be entered in either uppercase or lowercase. With MS-DOS versions 3.0 and later, external commands can be preceded by a drive and/or path.



REPLACE

REPLACE 3.2
Update Files External

Purpose

Selectively adds or replaces files on a disk.

Syntax

REPLACE [*drive:*]*pathname* [*drive:*][*path*] [/A][/D][/P][/R][/S][/W]

where:

pathname	is the name and location of the source files to be transferred, optionally preceded by a drive; wildcard characters are permitted in the filename.
drive:path	is the destination for the file being transferred; filenames are not permitted in the destination parameter.
/A	transfers only those source files that do not exist at the destination (cannot be used with /S or /D).
/D	transfers only those source files with a more recent date than their destination counterparts (cannot be used with /A).
/P	prompts the user for confirmation before each file is transferred.
/R	allows REPLACE to overwrite destination read-only files.
/S	searches all subdirectories of the destination directory for a match with the source files (cannot be used with /A).
/W	causes REPLACE to wait for the disk to be changed before transferring files.

Description

The REPLACE utility allows files to be updated easily to more recent versions. REPLACE examines the source and destination directories and, depending on the switches used in the command line, selectively updates matching files or copies only those files that exist on the source disk but not the destination disk.

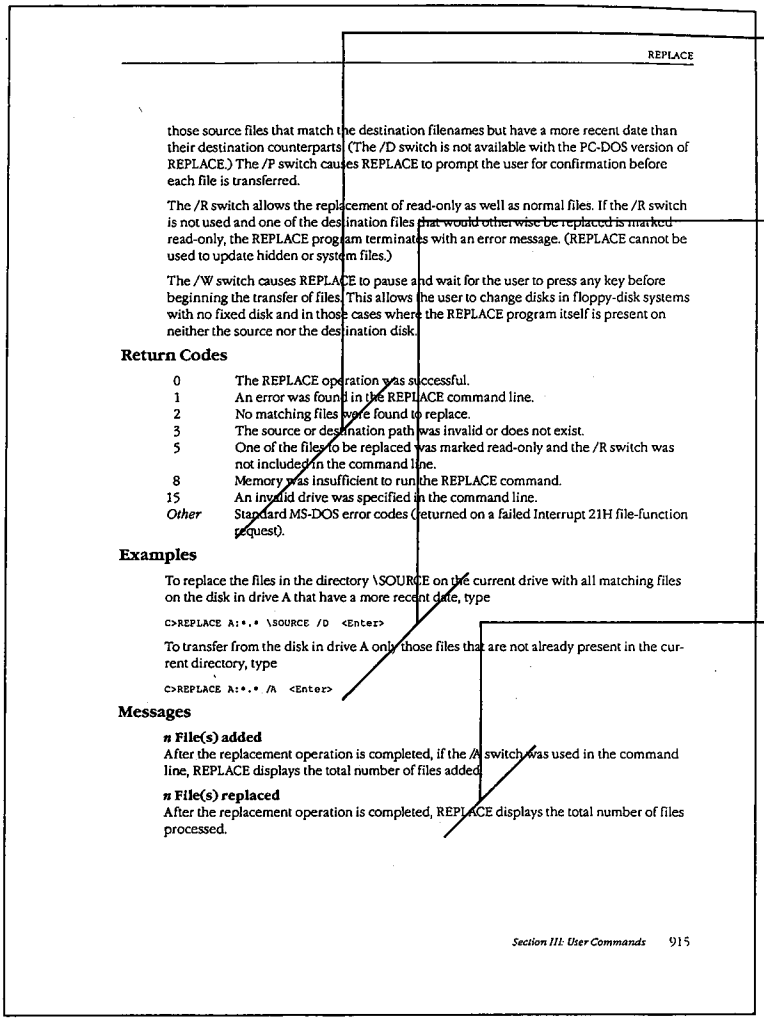
The *pathname* parameter (the source) specifies the name and location of the files to be transferred (optionally preceded by a drive); wildcards are permitted in the filename. The *drive:path* parameter (the destination) specifies the location of the files to be replaced and can consist of a drive, a path, or both. If only a drive is specified as the destination, REPLACE assumes the current directory of the disk in that drive. If the destination is omitted completely, REPLACE assumes the current drive and directory. The /S switch causes REPLACE to also search all subdirectories of the destination directory for files to be replaced.

The /A, /D, and /P switches allow selective replacement of files on the destination disk. When the /A switch is used, REPLACE transfers only those files on the source disk that do not exist in the destination directory. When the /D switch is used, REPLACE transfers only

914 *The MS-DOS Encyclopedia*

BELOW WHERE
A brief explanation of each command parameter and switch. Drives, paths, and filenames are always listed first, followed by the switches in alphabetic order. Any special position required for a filename or switch is shown in the syntax line and noted in the explanation.

DESCRIPTION
A detailed description of the command, including a full explanation of MS-DOS version dependencies, default values, possible interactions of command parameters and options, useful background information, and any applicable warnings.



REPLACE

those source files that match the destination filenames but have a more recent date than their destination counterparts. (The /D switch is not available with the PC-DOS version of REPLACE.) The /P switch causes REPLACE to prompt the user for confirmation before each file is transferred.

The /R switch allows the replacement of read-only as well as normal files. If the /R switch is not used and one of the destination files that would otherwise be replaced is marked read-only, the REPLACE program terminates with an error message. (REPLACE cannot be used to update hidden or system files.)

The /W switch causes REPLACE to pause and wait for the user to press any key before beginning the transfer of files. This allows the user to change disks in floppy-disk systems with no fixed disk and in those cases where the REPLACE program itself is present on neither the source nor the destination disk.

Return Codes

0	The REPLACE operation was successful.
1	An error was found in the REPLACE command line.
2	No matching files were found to replace.
3	The source or destination path was invalid or does not exist.
5	One of the files to be replaced was marked read-only and the /R switch was not included in the command line.
8	Memory was insufficient to run the REPLACE command.
15	An invalid drive was specified in the command line.
Other	Standard MS-DOS error codes (returned on a failed Interrupt 21H file-function request).

Examples

To replace the files in the directory \SOURCE on the current drive with all matching files on the disk in drive A that have a more recent date, type

```
C>REPLACE A:*.* \SOURCE /D  <Enter>
```

To transfer from the disk in drive A only those files that are not already present in the current directory, type

```
C>REPLACE A:*.* /A  <Enter>
```

Messages

***n* File(s) added**
After the replacement operation is completed, if the /A switch was used in the command line, REPLACE displays the total number of files added.

***n* File(s) replaced**
After the replacement operation is completed, REPLACE displays the total number of files processed.

Section III: User Commands 915

RETURN CODES
Exit codes returned by the command (if any) that can be tested in a batch file or by another program.

EXAMPLES
One or more examples of the command at work, including examples of the resulting output where appropriate. User entry appears in color; do not type the prompt, which appears in black. Press the Enter key (labeled Return on some keyboards) as directed at the end of each command line.

MESSAGES
An alphabetic list of messages that may be displayed when the command is used in MS-DOS version 3.2 (may vary slightly in earlier versions). Both messages generated by the command itself and applicable messages generated by MS-DOS are included. Following each message is a brief explanation of the condition that produces the message and, where appropriate, any action that should be taken.

Contents by Functional Group

The MS-DOS commands can be divided into several distinct groups according to the functions they perform. These are listed on the following pages.

Command	Action
System Configuration and Control	
BREAK	Set Control-C check.
COMMAND	Install secondary copy of command processor.
DATE	Set date.
EXIT	Terminate command processor.
PROMPT	Define system prompt.
SELECT	Configure system disk for a specific country.
SET	Set environment variable.
SHARE	Install file-sharing support.
TIME	Set system time.
VER	Display version.
Character-Device Management	
CLS	Clear screen.
CTTY	Assign standard input/output.
GRAFTABL	Load graphics character set.
GRAPHICS	Print graphics screen-dump program.
KEYB*xx*	Define keyboard.
MODE	Configure device.
PRINT	Print file (background print spooler).
File Management	
ATTRIB	Change file attributes.
BACKUP	Back up files.
COMP	Compare files.
COPY	Copy file or device.
DEL/ERASE	Delete file.
EDLIN	Create or modify text file (see also commands below).
FC	Compare files.
RECOVER	Recover files.
RENAME	Change filename.
REPLACE	Update files.
RESTORE	Restore backup files.
TYPE	Display file.
XCOPY	Copy files.

(more)

Command	Action
Filters	
FIND	Find string.
MORE	Display by screenful.
SORT	Sort file or character stream alphabetically.
Directory Management	
APPEND	Set data-file search path.
CHDIR	Change current directory.
DIR	Display directory.
MKDIR	Make directory.
PATH	Define command search path.
RMDIR	Remove directory.
TREE	Display directory structure.
Disk Management	
ASSIGN	Assign drive alias.
CHKDSK	Check disk status.
DISKCOMP	Compare floppy disks.
DISKCOPY	Copy floppy disks.
FORMAT	Initialize disk.
FDISK	Configure fixed disk.
JOIN	Join disk to directory.
LABEL	Display volume label.
SUBST	Substitute drive for subdirectory.
SYS	Transfer system files.
VERIFY	Set verify flag.
VOL	Display disk name.
Installable Device Drivers	
ANSI.SYS	ANSI console driver.
DRIVER.SYS	Configurable external-disk-drive driver.
RAMDRIVE.SYS	Virtual disk.
VDISK.SYS	Virtual disk.
System-Configuration File Directives	
BREAK	Configure Control-C checking.
BUFFERS	Configure internal disk buffers.
COUNTRY	Set country code.
DEVICE	Install device driver.
DRIVPARM	Set block-device parameters.
FCBS	Set maximum open files using File Control Blocks (FCBs).

(more)

Command	Action
System-Configuration File Directives *(continued)*	
FILES	Set maximum open files using handles.
LASTDRIVE	Set highest logical drive.
SHELL	Specify command processor.
STACKS	Configure internal stacks.
Batch-File Directives	
AUTOEXEC.BAT	System startup batch file.
ECHO	Display text.
FOR	Execute command on file set.
GOTO	Jump to label.
IF	Perform conditional execution.
PAUSE	Suspend batch-file execution.
REM	Include comment line.
SHIFT	Shift replaceable parameters.
EDLIN Commands	
linenumber	Edit line.
A	Append lines from disk.
C	Copy lines.
D	Delete lines.
E	End editing session.
I	Insert lines.
L	List lines.
M	Move lines.
P	Display in pages.
Q	Quit.
R	Replace text.
S	Search for text.
T	Transfer another file.
W	Write lines to disk.

ANSI.SYS

ANSI Console Driver

2.0 and later

External

Purpose

Allows the user to employ a subset of the American National Standards Institute (ANSI) standard escape sequences for control of the console.

Syntax

DEVICE=[*drive*:][*path*]ANSI.SYS

where:

drive:*path* is the drive and/or path to search for ANSI.SYS if it is not in the root directory of the startup disk.

Description

The ANSI.SYS file contains an installable character-device driver that supersedes the system's default driver for the console device (video display and keyboard). After ANSI.SYS is installed by means of a *DEVICE=ANSI.SYS* command in the CONFIG.SYS file of the disk used to start the system, programs can use a subset of the ANSI 3.64-1979 standard escape sequences to erase the display, set the display mode and attributes, and control the cursor in a hardware-independent fashion. (A supplementary set of escape sequences that are not part of the ANSI standard allows reprogramming of the keyboard.)

Programs that use ANSI.SYS for control of the screen can run on any MS-DOS machine without modification, regardless of its hardware configuration. However, most popular application programs for the IBM PC and compatibles circumvent ANSI.SYS and manipulate the video controller and its video buffer directly to achieve maximum performance.

The ANSI.SYS device driver detects ANSI escape sequences in a character stream and interprets them as commands to control the keyboard and display. An ANSI escape sequence is a sequence of ASCII characters, the first two of which must be the Escape character (1BH) and the left-bracket character (5BH). The characters following the Escape and left-bracket characters vary with the type of control function being performed; most consist of an alphanumeric code followed by a letter. In some cases this code is a single character; in others it is more than one character or a two-part string separated by a semicolon. Each ANSI escape sequence ends in a unique letter character that identifies the sequence; case is significant for these letters. The escape sequences supported by the ANSI.SYS driver are summarized in the tables on the following pages.

An escape sequence cannot be entered directly at the system prompt because each ANSI escape sequence must begin with an Escape character, and pressing the Esc key (or Alt-27 on the numeric keypad) causes MS-DOS to cancel the command line. There are three methods of executing ANSI escape sequences that do not require writing a program:

- Include the escape sequences in a PROMPT command.
- Enter the escape sequences into a word processor or text editor, save the file as an ASCII text file, and then execute the file by using the TYPE or COPY command (specifying CON as the destination for COPY) from the MS-DOS system prompt.
  (If the escape sequences are echoed on the screen when the file is executed, a *DEVICE=ANSI.SYS* command was not included in the CONFIG.SYS file when the system was turned on.)
- Place the escape sequences in a batch (.BAT) file as part of an ECHO command. When the batch file is executed, the sequences are sent to the console.

When escape sequences are entered using the PROMPT command, the Escape character is entered as $e. When escape sequences are entered using a word processor to create an ASCII text or batch file, the Escape character is usually entered by pressing the Esc key or by holding down the Alt key while typing 27 on the numeric keypad. (See the documentation provided with the word-processor for specific instructions.) In most cases, the escape character will appear in the word processor or text editor as a back-arrow character (←) or a caret–left bracket combination (^[).

Note: When the escape character is represented as ^[(as it is in EDLIN, for example), an additional left-bracket character must still be added to properly begin an ANSI escape sequence. Thus, the beginning of a valid ANSI escape sequence in EDLIN appears as ^[[.

The tables in this section use the abbreviation ESC to show where the ASCII escape character 27 (1BH) appears in the string.

Note: Case is significant for the terminal character in the string.

The following escape sequences control cursor movement:

Operation	Escape Sequence	Effect
Cursor Up	ESC[*number*A	Moves the cursor up *number* rows (1–24, default = 1). Has no effect if cursor is on the top row.
Cursor Down	ESC[*number*B	Moves the cursor down *number* rows (1–24, default = 1). Has no effect if cursor is on the bottom row.
Cursor Right	ESC[*number*C	Moves the cursor right *number* rows (1–79, default = 1). Has no effect if cursor is in the far right column.
Cursor Left	ESC[*number*D	Moves the cursor left *number* rows (1–79, default = 1). Has no effect if cursor is in the far left column.
Position Cursor	ESC[*row*;*column*H	Moves the cursor to the specified row (1–25, default = 1) and column (1–80, default = 1). If *row* is omitted, the semicolon before *column* must be specified.

(more)

Operation	Escape Sequence	Effect
Position Cursor	ESC[*row;columnf*	Same as above.
Save Cursor Position	ESC[s	Stores the current row and column position of the cursor. Cursor can be restored to this position later with a Restore Cursor Position escape sequence.
Restore Cursor Position	ESC[u	Moves the cursor to the position of the most recent Save Cursor Position escape sequence.

The following two escape sequences are used to erase all or part of the display:

Operation	Escape Sequence	Effect
Erase Display	ESC[2J	Clears the screen and places the cursor at the home position.
Erase Line	ESC[K	Erases from the cursor position to the end of the same row.

The following escape sequences control the width and the color capability of the display. The use of any of these sequences clears the screen.

Operation	Escape Sequence	Effect
Set Mode	ESC[=0h	Sets display to 40 x 25 monochrome (text).
	ESC[=1h	Sets display to 40 x 25 color (text).
	ESC[=2h	Sets display to 80 x 25 monochrome (text).
	ESC[=3h	Sets display to 80 x 25 color (text).
	ESC[=4h	Sets display to 320 x 200 4-color (graphics).
	ESC[=5h	Sets display to 320 x 200 4-color (graphics, color burst disabled).
	ESC[=6h	Sets display to 640 x 200 2-color (graphics).

The following escape sequences control whether characters will wrap around to the first column of the next row after the rightmost column in the current row has been filled:

Operation	Escape Sequence	Effect
Enable Character Wrap	ESC[=7h	Sets character wrap.
Disable Character Wrap	ESC[=7l	Disables character wrap. (Note that the terminating letter is a lowercase L.)

The following escape sequence controls specific graphics attributes such as intensity, blinking, superscript, and subscript, as well as the foreground and background colors:

ESC[*attrib*;...;*attrib*m

where:

attrib is one or more of the following values. Multiple values must be separated by semicolons.

Value	Attribute	Value	Foreground Color	Value	Background Color
0	All attributes off	30	Black	40	Black
1	High intensity (bold)	31	Red	41	Red
2	Normal intensity	32	Green	42	Green
4	Underline (monochrome only)	33	Yellow	43	Yellow
5	Blink	34	Blue	44	Blue
7	Reverse video	35	Magenta	45	Magenta
8	Concealed (invisible)	36	Cyan	46	Cyan
		37	White	47	White

Note: Values 30 through 47 meet the ISO 6429 standard.

The following escape sequence allows redefinition of keyboard keys to a specified *string*:

ESC[*code*;*string*;...p

where:

code is one or more of the following values that represent keyboard keys. Semicolons shown in this table must be entered in addition to the required semicolons in the command line.

string is either the ASCII code for a single character or a string contained in quotation marks. For example, both 65 and "A" can be used to represent an uppercase A.

Key	Code			
	Alone	Shift-	Ctrl-	Alt-
F1	0;59	0;84	0;94	0;104
F2	0;60	0;85	0;95	0;105
F3	0;61	0;86	0;96	0;106
F4	0;62	0;87	0;97	0;107
F5	0;63	0;88	0;98	0;108
F6	0;64	0;89	0;99	0;109

(more)

Key	Code			
	Alone	**Shift-**	**Ctrl-**	**Alt-**
F7	0;65	0;90	0;100	0;110
F8	0;66	0;91	0;101	0;111
F9	0;67	0;92	0;102	0;112
F10	0;68	0;93	0;103	0;113
Home	0;71	55	0;119	–
Up Arrow	0;72	56	–	–
Pg Up	0;73	57	0;132	–
Left Arrow	0;75	52	0;115	–
Down Arrow	0;77	54	0;116	–
End	0;79	49	0;117	–
Down Arrow	0;80	50	–	–
Pg Dn	0;81	51	0;118	–
Ins	0;82	48	–	–
Del	0;83	46	–	–
PrtSc	–	–	0;114	–
A	97	65	1	0;30
B	98	66	2	0;48
C	99	67	3	0;46
D	100	68	4	0;32
E	101	69	5	0;18
F	102	70	6	0;33
G	103	71	7	0;34
H	104	72	8	0;35
I	105	73	9	0;23
J	106	74	10	0;36
K	107	75	11	0;37
L	108	76	12	0;38
M	109	77	13	0;50
N	110	78	14	0;49
O	111	79	15	0;24
P	112	80	16	0;25
Q	113	81	17	0;16
R	114	82	18	0;19
S	115	83	19	0;31
T	116	84	20	0;20
U	117	85	21	0;22
V	118	86	22	0;47
W	119	87	23	0;17
X	120	88	24	0;45

(more)

Key	Code			
	Alone	**Shift-**	**Ctrl-**	**Alt-**
Y	121	89	25	0;21
Z	122	90	26	0;44
1	49	33	–	0;120
2	50	64	–	0;121
3	51	35	–	0;122
4	52	36	–	0;123
5	53	37	–	0;124
6	54	94	–	0;125
7	55	38	–	0;126
8	56	42	–	0;127
9	57	40	–	0;128
0	48	41	–	0;129
–	45	95	–	0;130
=	61	43	–	0;131
Tab	9	0;15	–	–
Null	0;3	–	–	–

Examples

The following examples use ESC or $e to show where the ASCII escape character 27 (1BH) appears in the string. The PROMPT examples can be typed as shown, but for the examples that use ESC to denote the escape character, the actual escape character should be typed in its place.

To move the cursor to row 10, column 30 and display the string *Main Menu*, use the escape sequence

```
ESC[10;30fMain Menu
```

or

```
ESC[10;30HMain Menu
```

To move the cursor to row 5, column 10 and display the letter A (*ESC[5;10fA*), move the cursor down one row (*ESC[B*), move the cursor back one space and display the letter B (*ESC[DB*), move the cursor down one row (*ESC[B*), and move the cursor back one space and display the letter C (*ESC[DC*), use the escape sequence

```
ESC[5;10fAESC[BESC[DBESC[BESC[DC
```

To use ANSI escape sequences with the PROMPT command to save the current cursor position (*$e[s*), move the cursor to row 1, column 69 (*$e[1;69f*), display the current time using the PROMPT command's $t function, restore the cursor position (*$e[u*), and then

display the current path using the PROMPT command's $p function and display a greater-than sign using the PROMPT command's $g function, use the escape sequence

```
C>PROMPT $e[s$e[1;69f$t$e[u$p$g  <Enter>
```

To erase the display (*ESC[2J*), then move the cursor to row 10, column 30 and display the string *Main Menu* (*ESC[10;30fMain Menu*), use the escape sequence

```
ESC[2JESC[10;30fMain Menu
```

To move the cursor to row 5, column 40 (*ESC[5;40f*) and erase the remainder of the row starting at the current cursor position (*ESC[K*), use the escape sequence

```
ESC[5;40fESC[K
```

To move the cursor to row 3 (*ESC[3;f*), erase the entire row (*ESC[K*), move the cursor down one row (*ESC[B*), erase that entire row (*ESC[K*), move the cursor down one row and erase that entire row, use the escape sequence

```
ESC[3;fESC[KESC[BESC[KESC[BESC[K
```

To set the display mode to 25 rows of 80 columns in color (*ESC[=3h*) and disable character wrap (*ESC[=7l*), use the escape sequence

```
ESC[=3hESC[=7l
```

Note that ESC[=3h will also clear the screen.

To enable character wrap, use the escape sequence

```
ESC[=7h
```

To set the foreground color to black and the background color to blue (*ESC[30;44m*), clear the display (*ESC[2J*), then position the cursor at row 10, column 30 and display the string *Main Menu* (*ESC[10;30fMain Menu*), use the escape sequence

```
ESC[30;44mESC[2JESC[10;30fMain Menu
```

To (effectively) exchange the backslash and question-mark keys using literal strings to denote the keys, use the escape sequence

```
ESC["\";"?"pESC["?";"\"p
```

To exchange the backslash and question-mark keys using each key's ASCII value to denote the key, use the escape sequence

```
ESC[92;63pESC[63;92p
```

To restore the backslash and question-mark keys to their original meanings, use the escape sequence

```
ESC["\";"\"pESC["?";"?"p
```

or

```
ESC[92;92pESC[63;63p
```

To redefine the Alt-F9 key combination (*ESC[0;112*) so that it issues a CLS command (*;"CLS"*) plus a carriage return (*;13*) to execute the CLS command, then issues a DIR command piped through the SORT filter starting at column 24 (*;"DIR | SORT /+24"*) followed by another carriage return, use the escape sequence

```
ESC[0;112;"CLS";13;"DIR | SORT /+24";13p
```

To restore the Alt-F9 key combination to its original meaning, use the escape sequence

```
ESC[0;112;0;112p
```

APPEND

Set Data-File Search Path

3.2

External

Purpose

Specifies a search path for open operations on data files. (Also supported with some implementations of version 3.1, for use with networks.)

Syntax

APPEND [[*drive*:]*path*] [;[*drive*:]*path* ...]

or

APPEND ;

where:

path is the name of a valid directory, optionally preceded by a drive.

Description

APPEND is a terminate-and-stay-resident program that is used to specify a path or paths to be searched for data files (in contrast with the PATH command, which specifies a path to be searched for executable or batch files). The search path can include a network drive. If a program attempts to open a file and the file is not found in the current or specified directory, each path given in the APPEND command is searched.

If the APPEND command is entered with a path consisting of only a semicolon character (;), a "null" search path for data files is set; that is, no directory other than the current or specified directory is searched. This effectively cancels any search paths previously set with an APPEND command but does not free the memory used by APPEND.

An APPEND command without any parameters displays the current search path(s) for data files.

Note that a program cannot detect whether an opened file was found where it was expected (in the current or specified directory) or in some other directory specified in the APPEND command.

Warning: When an assigned drive is to be part of the search path, the ASSIGN command must be used before the APPEND command. Use of the ASSIGN command should be avoided whenever possible because it hides drive characteristics from those programs that require detailed knowledge of the drive size and format.

Examples

To cause the directories C:\SYSTEM and C:\SOURCE to be searched for a file during an open operation if the file is not found in the current or specified directory, type

```
C>APPEND C:\SYSTEM;C:\SOURCE  <Enter>
```

To display the current search path for data files, type

```
C>APPEND  <Enter>
```

MS-DOS then displays

```
APPEND=C:\SYSTEM;C:\SOURCE
```

To ensure that no directories other than the current or specified directory are searched during a file open operation, type

```
C>APPEND ;  <Enter>
```

Messages

APPEND / ASSIGN Conflict
APPEND was used before ASSIGN.

Incorrect DOS version
The version of APPEND is not compatible with the version of MS-DOS that is running.

No appended directories
The APPEND command had no parameters and no APPEND search path is active.

ASSIGN

Assign Drive Alias

3.0 and later

External

Purpose

Redirects requests for disk operations on one drive to a different drive. (Available with PC-DOS beginning with version 2.0.)

Syntax

ASSIGN [*x*=*y* [...]]

where:

x	is a valid designator (A, B, C, etc.) for a disk drive that physically exists in the system.
y	is a valid designator for the drive to be accessed by references to *x*.

Description

ASSIGN is a terminate-and-stay-resident program that redirects all references to drive *x* or files on drive *x* to drive *y*. The ASSIGN command is intended for use with application programs that require files to reside on drive A or B and have no provision within the program for changing those drives.

Multiple drive assignments can be requested in the same ASSIGN command line; the drive pairs must be separated with spaces, commas, or semicolons. Unlike the form in most other MS-DOS commands, the drive letters are not followed by colon characters (:). When a single drive is assigned, the equal sign is optional.

ASSIGN commands are not incremental. Each new ASSIGN command replaces assignments made with the previous ASSIGN command and cancels any assignments not specifically replaced. Entering ASSIGN with no parameters cancels all current drive assignments.

Warning: Use of the ASSIGN command should be avoided whenever possible because it hides drive characteristics from those programs that require detailed knowledge of the drive size and format; in particular, drives redirected with an ASSIGN statement should never be used with a BACKUP, RESTORE, LABEL, JOIN, SUBST, or PRINT command. ASSIGN can also defeat the checking performed by the COPY command to prevent a file from being copied onto itself. The FORMAT, SYS, DISKCOPY, and DISKCOMP commands ignore any drive reassignments made with ASSIGN.

With MS-DOS versions 3.1 and later, the SUBST command should be used instead of ASSIGN. For example, the command

```
C>ASSIGN A=C  <Enter>
```

should be replaced with the command

```
C>SUBST A: C:\  <Enter>
```

Examples

To redirect all requests for drive A to drive C, type

```
C>ASSIGN A=C  <Enter>
```

To redirect all requests for drives A and B to drive C, type

```
C>ASSIGN A=C B=C  <Enter>
```

To cancel all drive redirections currently in effect, type

```
C>ASSIGN  <Enter>
```

Messages

Incorrect DOS version
The version of ASSIGN is not compatible with the version of MS-DOS that is running.

Invalid parameter
One of the specified drive designators refers to a drive that does not exist in the system.

ATTRIB

Change File Attributes

3.0 and later

External

Purpose

Sets, removes, or displays a file's read-only and/or archive attributes.

Syntax

ATTRIB [+R¦–R] [+A ¦–A] [*drive*:]*pathname*

where:

+R	marks the file read-only.
–R	removes the read-only attribute.
+A	sets the file's archive flag (version 3.2).
–A	removes the file's archive flag (version 3.2).
pathname	is the name and location, optionally preceded by a drive, of the file whose attributes are to be changed or displayed; wildcard characters are permitted in the filename.

Description

Each file has an entry in the disk's directory that contains its name, location, and size; the date and time it was created or last modified; and an attribute byte. For normal files, bits 0, 1, 2, and 5 in the attribute byte designate, respectively, whether the file is read-only, hidden, or system and whether it has been changed since it was last backed up.

The ATTRIB command provides a way to alter the read-only and archive bits from the MS-DOS command level. If a file is marked read-only, it cannot be deleted or modified; thus, crucial programs or data can be protected from accidental erasure. A file's archive flag can be used together with the /M switch of the BACKUP command or the /M or /A switch of the XCOPY command to allow an incremental or selective backup of files from one disk to another.

If the ATTRIB command is entered with only a pathname, the current attributes of the selected file are displayed. An R is displayed next to the name of a file that is marked read-only and an A is displayed if the file has the archive flag set.

Examples

To make the file MENUMGR.C in the current directory of the current drive a read-only file, type

```
C>ATTRIB +R MENUMGR.C  <Enter>
```

To display the attributes of the file LETTER.DOC in the directory \SOURCE on the disk in drive D, type

```
C>ATTRIB D:\SOURCE\LETTER.DOC  <Enter>
```

MS-DOS then displays

```
R A       D:\SOURCE\LETTER.DOC
```

to indicate that the file is marked read-only and the archive flag has been set.

To set the archive flag on all files in the directory \SYSTEM on drive C and mark them as read-only, type

```
C>ATTRIB +A +R C:\SYSTEM\*.*  <Enter>
```

Messages

Access denied
ATTRIB cannot be used to alter or replace the attributes of a file in use across a network.

DOS 2.0 or later required
ATTRIB does not work with versions of MS-DOS earlier than 2.0.

Incorrect DOS version
The version of ATTRIB is not compatible with the version of MS-DOS that is running.

Invalid number of parameters
More than two attributes were used before the pathname.

Invalid path or file not found
The file named in the command line or one of the directories in the given path does not exist.

Syntax error
An invalid attribute was supplied or the attribute was not properly placed before the pathname in the command line.

BACKUP

Back Up Files

2.0 and later

External

Purpose

Creates backup copies of files, along with the associated directory information necessary to restore the files to their original locations.

Syntax

BACKUP *source destination* [/A] [/D:*date*] [/L:*filename*] [/M] [/P] [/S] [/T:*time*]

where:

source	is the location (drive and/or path) and, optionally, the name of the files to be backed up; wildcard characters are permitted in the filename.
destination	is the drive to receive the backup files.
/A	adds the files to existing files on the destination disk without erasing the destination disk.
/D:*date*	backs up only those files modified on or after *date*.
/L:*filename*	creates a log file with the specified name in the root directory of the disk being backed up. If *filename* is not specified, BACKUP creates a file named BACKUP.LOG and places the log entries there. Use of the /L:*filename* switch may cause loss of IBM compatibility.
/M	backs up only those files modified since the last backup.
/P	packs the destination disk with as many files as possible, creating subdirectories, if necessary, to hold some of the files. Use of the /P switch causes loss of IBM compatibility.
/S	backs up the contents of all subdirectories of the source directory.
/T:*time*	backs up only those files modified on or after *time*.

Note: Not all switches are supported by all implementations of MS-DOS.

Description

The BACKUP command creates a backup copy of the specified file or files, transferring them from either a floppy disk or a fixed disk to another removable or fixed disk. The backup file is in a special format that includes information about the original file's location in the directory structure. Files created by BACKUP can be restored to their original form only with the RESTORE command.

BACKUP can back up a single file or many files in the same operation. If only a drive letter is given as the source, all the files in the current directory of that disk are backed up. If only a path is given as the source, all the files in the specified directory are backed up. If the /S switch is used, all the files in the current or specified directory are backed up, and

the files in all its subdirectories as well. If both a path and a filename are entered as the source, the specified file or files in the named directory are backed up.

If the source file is marked read-only, the resulting backup file will also be marked read-only. If the source file's archive bit is set, it will be cleared for both the source and the destination files. BACKUP also backs up hidden files; the files will remain hidden on the destination disk.

If the destination disk is a floppy disk, its previous contents are erased as part of the backup operation (unless the /A switch is included in the command line and the destination disk has already been used as a backup disk—that is, the disk contains a valid BACKUPID.@@@ file). If the files being backed up do not fit onto a single floppy disk, the user will be prompted to insert additional disks until the backup operation is complete.

If the destination disk is a fixed disk, the backed-up files are placed in a directory named \BACKUP. If a \BACKUP directory already exists on the fixed disk, any files previously contained in it are erased as part of the backup operation (unless the /A switch is included in the command line and the destination disk has already been used as a backup disk—that is, the \BACKUP directory contains a valid BACKUPID.@@@ file). Other files on the destination fixed disk are not disturbed.

A control file named BACKUPID.@@@ is placed on every floppy disk onto which files are backed up or in the /BACKUP directory if the files are backed up onto a fixed disk. The BACKUPID.@@@ file has the following format:

Byte	Value	Use
00H	00 or FFH	Not last floppy disk/last floppy disk
01–02H	*nn*	Floppy disk number in low-byte/high-byte decimal format
03–04H	*nnnn*	Full year in low-byte/high-byte order
05H	1–31	Day of the month
06H	1–12	Month of the year
07–0AH	*nnnn*	Standard MS-DOS system time if the /T:*time* switch was used; otherwise 0
0B–7FH	00	Not used

Each backed-up file also has a 128-byte header added to it when it is created. The header has the following format:

Byte	Value	Use
00H	00 or FFH	Not last floppy disk/last floppy disk on which this file resides
01H	*nn*	Floppy disk number
02–04H	00	Not used

(more)

Byte	Value	Use
05–44H	*nn*	File's full pathname, except for drive designator
45–52H	00	Not used
53H	*nn*	Length of the file's pathname plus one
54–7FH	00	Not used

The /T:*time*, /D:*date*, and /M switches allow incremental or partial backups. The /T:*time* switch excludes files modified or created before a certain time and should be used in the form of the COUNTRY command in effect. For the USA, the format is /T:*hh:mm:ss*. (The /T:*time* switch is not supported in all implementations of BACKUP.) The /D:*date* switch excludes files modified or created before a certain date and should be used in the form of the COUNTRY command in effect. For the USA, the format is /D:*mm-dd-yy*. The /M switch selects only those files that have been modified since the last backup operation.

The /L:*filename* switch causes a log file to be created on the source disk. This file includes the name of each file backed up, the time and date, and the number of the destination disk that received that backup file. If *filename* is omitted, the name defaults to BACKUP.LOG. Use of the /L:*filename* switch can cause compatibility problems between MS-DOS and PC-DOS because the backup log file may match the search pattern and be backed up, too, resulting in an extra file on the backup disk.

The /P switch causes backup files to be packed as densely as possible on the destination disk. When many short files are being backed up to floppy disks, the number of files that fit on the destination disk may exceed the number of entries that will fit in the destination's root directory. If the /P switch is included in the command line, subdirectories are created on the destination disk as needed to use the disk space more effectively. The /P switch is not supported under PC-DOS; backup disks created with the /P switch will not be compatible with IBM's BACKUP and RESTORE commands.

Warning: BACKUP should not be used on disk directories or drives that have been redirected with an ASSIGN, JOIN, or SUBST command.

Return Codes

0 Backup operation was successful.
1 No files were found to back up.
2 Some files were not backed up because of sharing conflicts (versions 3.0 and later).
3 Backup operation was terminated by user.
4 Backup operation was terminated because of error.

Examples

To back up the file REPORT.TXT in the current directory on the current drive, placing the backup file on the disk in drive A, type

```
C>BACKUP REPORT.TXT A:  <Enter>
```

To back up all the files in the subdirectory B:\V2\SOURCE, placing the backup files on the disk in drive A, type

```
C>BACKUP B:\V2\SOURCE A:  <Enter>
```

To back up all the files with extension .C in the directory \V2\SOURCE on the current drive, placing the backup files on the disk in drive A, type

```
C>BACKUP \V2\SOURCE\*.C A:  <Enter>
```

To back up all the files with the extension .ASM from the current directory on the current drive and from all its subdirectories, placing the backup files on the disk in drive A, type

```
C>BACKUP *.ASM A: /S  <Enter>
```

To back up all the files that have been modified since the last backup from all the subdirectories on drive C, placing the backup files on the disk in drive A, type

```
C>BACKUP C:\ A: /S /M  <Enter>
```

To back up all the files with the extension .C from the directory C:\V2\SOURCE that were modified on or after October 16, 1985, placing the backup files on the disk in drive A, type

```
C>BACKUP C:\V2\SOURCE\*.C A: /D:10-16-85  <Enter>
```

Messages

*****Backing up files to drive *X*: *****
Diskette Number: *n*
This informational message informs the user of the progress of the BACKUP command.

*****Last file not backed up *****
The destination drive does not have enough space to back up the last file.

*****Not able to back up file *****
One of the system calls used by BACKUP failed unexpectedly; for example, a file could not be opened, read, or written.

Cannot create Subdirectory BACKUP on drive *X*:
Drive *X* is full or its root directory is full.

DOS 2.0 or later required
BACKUP does not work with versions of MS-DOS earlier than 2.0.

Error trying to open backup log file
Continuing without making log entries
The /L switch was used and BACKUP is unable to create the backup log file.

Files cannot be added to this diskette
unless the PACK (/P) switch is used
Set the switch (Y/N)?
The root directory of the destination disk is full and a subdirectory must be created to hold the remaining files. Respond with *Y* to cause BACKUP to create a subdirectory and continue backing up files into it; respond with *N* to return to MS-DOS.

Incorrect DOS version
The version of BACKUP is not compatible with the version of MS-DOS that is running.

Insert backup diskette in drive *X*:
Strike any key when ready
This message prompts the user to insert a disk to receive the backup files into the specified destination drive.

Insert backup diskette *n* in drive *X*:
Strike any key when ready
The files being backed up will not fit onto a single floppy disk; this message prompts the user to insert the next floppy disk. Multiple-floppy-disk backup disks should be labeled and numbered to match the number displayed in this message.

Insert backup source diskette in drive *X*:
Strike any key when ready
This message prompts the user to insert the floppy disk to be backed up into the specified source drive.

Insert last backup diskette in drive *X*:
Strike any key when ready
This message prompts the user to insert the final disk that will receive the backup files into the specified destination drive.

Insufficient memory
Available system memory is insufficient to run the BACKUP program.

Invalid argument
One of the switches specified in the command line is invalid or is not supported in the version of BACKUP being used.

Invalid Date/Time
An invalid date or time was given with the /D:*date* or /T:*time* switch.

Invalid drive specification
The source or destination drive specified in the command line is not available or is not valid.

Invalid number of parameters
At least two parameters, the source and the destination, must be specified in the command line; a maximum of seven switches can be specified after the source and destination.

Invalid parameter
One of the switches supplied in the command line is invalid.

Invalid path
The path specified as the source is invalid or does not exist.

Last backup diskette not inserted
Insert last backup diskette in drive *X*:
Strike any key when ready
The backup disk inserted as the last backup disk was not the correct disk. Insert the correct disk.

No space left on device
The destination disk is full.

No such file or directory
The source specified is invalid or does not exist.

Source and target drives are the same
The disks specified as the source and destination disks are identical.

Source disk is Non-removable
The disk containing the files to be backed up is a fixed disk.

Target can not be used for backup
The disk specified as the destination disk is damaged or the /A switch was used in the command line and the disk does not contain a valid BACKUPID.@@@ file.

Target disk is Non-removable
The disk that will contain the backed-up files is a fixed disk.

Target is a floppy disk

or

Target is a hard disk
This informational message indicates which type of disk was specified as the destination disk.

Too many open files
Too many files are open. Increase the value of the FILES command in the CONFIG.SYS file.

Unable to erase *filename*
BACKUP is unable to erase an older version of a backed-up file because the file is read-only or is in use by another program.

Warning! Files in the target drive
***X*:\root directory will be erased**
Strike any key when ready
The destination is a floppy-disk drive and this message warns the user that all files in its root directory will be erased before the backup operation.

Warning! Files in the target drive
C:\BACKUP directory will be erased
Strike any key when ready
BACKUP is ready to begin backing up files to the \BACKUP directory on drive C. All existing files in the \BACKUP directory will be deleted. Press Crtl-Break to terminate the backup operation or press any key to continue.

Warning! No files were found to back up
No files were found on the source disk in the current or specified directory or no files were found matching the filename supplied.

BATCH

1.0 and later

System Batch-File Interpreter

Internal

Purpose

Sequentially executes commands stored in a batch file (a text-only file with a .BAT extension).

Syntax

filename [[*parameter1* [*parameter2* [...]]]]

where:

filename	is the name of the batch file to be executed, without the .BAT extension. (The filename is always %0 in the list of replaceable parameters.)
parameter1	is the filename, switch, or string that is the value of the first replaceable parameter (%1).
parameter2	is the filename, switch, or string that is the value of the second replaceable parameter (%2). As many additional replaceable parameters can be specified as the command line will hold.

Description

A batch file is an ASCII text file that contains one or more MS-DOS commands. It is a useful way to perform sequences of frequently used commands without having to type them all each time they are needed. When a batch file is invoked by entering its name, the commands it contains are carried out in sequence by a special batch-file interpreter built into COMMAND.COM. Additional information entered in the batch-file command line can be passed to other programs by means of replaceable parameters (*see below*).

A batch file must always have the extension .BAT. The file can contain any number of lines of ASCII text; each line can contain a maximum of 128 characters. Batch files can be created with EDLIN or another text editor or with a word processor in nondocument mode. (Formatted document files cannot be used as batch files because they contain special control codes or escape sequences that cannot be processed by the batch-file interpreter.) Batch files can also be created with the MS-DOS COPY command by specifying the CON device (keyboard) as the source file and the desired batch-file name as the destination file. For example, after the command

```
C>COPY CON MYFILE.BAT  <Enter>
```

each line that is typed will be placed into MYFILE.BAT. This form of the COPY command is terminated by pressing Ctrl-Z or the F6 key, followed by the Enter key.

The commands in a batch file can be any combination of internal MS-DOS commands (such as DIR or COPY), external MS-DOS commands (such as CHKDSK or BACKUP), the names of other programs or batch files, or the following special batch-file directives:

Command	Action
ECHO	Displays a message on standard output (versions 2.0 and later).
FOR	Executes a command on each of a set of files (versions 2.0 and later).
GOTO	Transfers control to another point in a batch file (versions 2.0 and later).
IF	Conditionally executes a command based on the existence of a file, the equality of two strings, or the return code of a previously run program (versions 2.0 and later).
PAUSE	Waits for the user to press a key before executing the remainder of the batch file.
REM	Allows comment lines to be placed in batch files for internal documentation.
SHIFT	Provides access to more than 10 command-line parameters (versions 2.0 and later).

These special batch commands are discussed individually, with examples, in the following pages.

A batch file is executed by entering its name, without the .BAT extension, in response to the MS-DOS prompt. The system's command processor, COMMAND.COM, searches the current directory and then each directory named in the PATH environment variable for a file with the specified name and the extension .COM, .EXE, or .BAT, in that order. If a .COM or .EXE file is found, it is loaded into memory and receives control; if a .BAT file is found, it is assumed to be a text file and is passed to the batch-file interpreter. (If two files with the same name exist in the same directory, one with a .COM or .EXE extension and the other with a .BAT extension, it is not possible to execute the .BAT file—the .COM or .EXE file is always loaded instead.)

If the disk that contains a batch file is removed before all the commands in the batch file are executed, COMMAND.COM will prompt the user to replace the disk so that the batch file can be completed. Execution of a batch file can be terminated by pressing Ctrl-C or Ctrl-Break, causing COMMAND.COM to issue the message *Terminate batch job? (Y/N)*. If the user responds with *Y*, the batch file is abandoned and COMMAND.COM displays its usual prompt.

The input redirection (<), output redirection (> or >>), and piping (¦) characters have no effect when they are used in a command line that invokes a batch file. However, they can be used in individual command lines *within* the file.

Ordinarily, if a batch file includes the name of another batch file, control passes to the second batch file and never returns. That is, when the commands in the second batch file are completed, the batch-file interpreter terminates and any remaining commands in the first

batch file are not processed. However, a batch file can execute another batch file without itself being terminated by first loading a secondary copy of the system's command processor. To accomplish this, the first batch file must contain a command of the form

COMMAND /C *batch2*

where *batch2* is the name of the second batch file. When all the commands in the second batch file have been processed, the secondary copy of COMMAND.COM exits and the first batch file continues where it left off. (*See* USER COMMANDS: COMMAND for details on the use of the /C switch with COMMAND.COM.)

A batch file can be made more flexible by including replaceable parameters inside the file. A replaceable parameter takes the form *%n*, where *n* is a numeral in the range 0 through 9. Replaceable parameters simply hold places in the batch file for filenames or other information that the user will supply in the command line when the batch file is invoked.

When a batch file is interpreted and a command containing a replaceable parameter is encountered, the corresponding value specified in the batch-file command line is substituted for the replaceable parameter and the command is then executed. The %0 replaceable parameter is replaced by the name of the batch file itself; parameters %1 through %9 are replaced sequentially with the remaining values specified in the command line. If a replaceable parameter references a command-line entry that does not exist, the parameter is replaced with a null (zero-length) string.

For example, if the batch file MYBATCH.BAT contains the single line

```
COPY %1.COM %2.SAV
```

and is executed by entry of

```
C>MYBATCH FILE1 FILE2  <Enter>
```

the actual command that is carried out is

```
COPY FILE1.COM FILE2.SAV
```

(The SHIFT batch command makes it possible to use more than 10 replaceable parameters. *See* USER COMMANDS: BATCH:SHIFT)

An environment variable is a special case of a replaceable parameter. If the SET command is used in the form

SET *name=value*

to add an environment variable to the system's environment block, the string *value* will be substituted for the string *%name%* wherever the latter is encountered during the interpretation of a batch file. This capability is available only in versions 2.x, 3.1, and 3.2.

BATCH: AUTOEXEC.BAT

1.0 and later

System Startup Batch File

Description

The AUTOEXEC.BAT file is an optional batch file containing a series of MS-DOS commands that automatically execute when the system is turned on or restarted.

When the system's default command processor, COMMAND.COM, is first loaded, it looks in the root directory of the current drive for a file named AUTOEXEC.BAT. If AUTOEXEC.BAT is not found, COMMAND.COM prompts the user to enter the current time and date and then displays the MS-DOS copyright notice and command prompt. If AUTOEXEC.BAT is found, COMMAND.COM sequentially executes the commands within the file. No prompts to enter the time and date are issued unless the TIME and DATE commands are explicitly included in the batch file; no copyright notice is displayed.

Typical uses of the AUTOEXEC.BAT file include

- Running a program to set the system time and date from a real-time clock/calendar located on a multipurpose expansion board (IBM PC, PC/XT, or compatibles only)
- Using the MODE command to configure a serial port or to redirect printing
- Executing SET commands to configure environment variables
- Setting display colors on a color monitor (if the command *DEVICE=ANSI.SYS* has been included in the CONFIG.SYS file)
- Installing terminate-and-stay-resident (TSR) utilities
- Using the PATH command to tell COMMAND.COM where to find executable program files if they are not in the current drive and/or directory
- Defining a custom prompt using the PROMPT command
- Invoking an application program such as a database, spreadsheet, or word processor

A secondary copy of the command processor can also be loaded from within the AUTOEXEC.BAT file. If this copy of COMMAND.COM is loaded with the /P switch, it too searches for an AUTOEXEC.BAT file on the current drive and processes the file if it is found. This feature can be useful for performing special operations. For example, on very old PCs that are unable to start from a fixed disk, a secondary copy of the command processor can be used to make the fixed disk's copy of COMMAND.COM the copy used by the system from that point on (at the expense of some system memory). If the AUTOEXEC.BAT file containing the lines

```
C:
COMMAND C:\ /P
```

is stored on the floppy disk in drive A when the system is turned on or restarted, the first line of the file causes drive C to become the current drive; then the second line

permanently loads a secondary copy of COMMAND.COM from drive C and instructs COMMAND.COM to reload its transient portion from the root directory of drive C when necessary. This in turn triggers the execution of the AUTOEXEC.BAT file on the fixed disk to perform the actual system configuration. Because the transient part of COMMAND.COM will be reloaded from the fixed disk when necessary, rather than from the floppy disk, system performance is improved considerably.

Example

The following example illustrates several common uses of the AUTOEXEC.BAT file to configure the MS-DOS system at startup time. (The line numbers are included for reference and are not part of the actual file.)

```
1    ECHO OFF
2    SETCLOCK
3    PROMPT $p$g
4    MD D:\BIN
5    COPY C:\SYSTEM\*.* D:\BIN > NUL
6    PATH=D:\BIN;C:\WP\WORD;C:\MSC\BIN;C:\ASM
7    APPEND D:\BIN;C:\WP\WORD;C:\ASM
8    SET INCLUDE=C:\MSC\INCLUDE
9    SET LIB=C:\MSC\LIB
10   SET TMP=C:\TEMP
11   MODE COM1:9600,n,8,1,p
12   MODE LPT1:=COM1:
```

Line 1 causes the batch-file processor to operate silently; that is, the commands in the batch file are not displayed on the screen as they are executed.

Line 2 runs a utility program called SETCLOCK, which reads the current time and date from a real-time clock chip on a multifunction board and sets the system time and date accordingly.

Line 3 configures COMMAND.COM's user prompt so that it displays the current drive and directory.

Line 4 creates a directory named \BIN on drive D, which in this case is a RAMdisk that was created by an entry in the system's CONFIG.SYS file.

Line 5 copies all the programs in the \SYSTEM directory on drive C to the \BIN directory on drive D. The normal output of this COPY command is redirected to the NUL device—in effect, the output is thrown away—to avoid cluttering the screen.

Line 6 sets the search path for executable files and line 7 sets the search path for data files. Note that the RAMdisk directory D:\BIN is specified as the first directory in the PATH command; therefore, if the name of a program is entered and it cannot be found in the current directory, COMMAND.COM will look next in the directory D:\BIN. This strategy allows commonly used programs (in this example, the programs in the \SYSTEM directory that were copied into D:\BIN) to be located and loaded quickly.

Lines 8 through 10 add the environment variables INCLUDE, LIB, and TMP to the system's environment. These variables are used by the Microsoft C Compiler and the Microsoft Object Linker.

Line 11 configures the first serial communications port (COM1) and line 12 causes program output to the system's first parallel port (LPT1) to be redirected to the first serial port. This pair of commands allows a serial-interface Hewlett Packard LaserJet printer to be used as the system list device.

Note: Depending on the version of MS-DOS in use, some commands in this example may not be available or may support different options. See the individual command entries for more detailed information.

BATCH: ECHO

2.0 and later

Display Text

Internal

Purpose

Displays a message during the execution of a batch file and controls whether or not batch-file commands are listed on the screen as they are executed.

Syntax

ECHO [ON¦OFF¦*message*]

where:

ON	enables the display of all subsequent batch-file commands as they are executed.
OFF	disables the display of all subsequent batch-file commands as they are executed.
message	is a text string to be displayed on standard output.

Description

Each command line of a batch file is ordinarily displayed on the screen as it is executed. The ECHO command has a dual usage: to control the display of these commands and to display a message to the user.

ECHO is used with ON or OFF to enable or disable the display of commands during batch-file processing. If the ECHO command is used with no parameter, the current status of the batch processor's ECHO flag is displayed. Note that the ECHO flag is always forced on at the start of any batch-file processing, even if that batch file was invoked by another batch file.

The ECHO command is not limited to batch files; an ECHO command can also be issued at the command prompt. ECHO OFF entered at the command prompt prevents the prompt from subsequently being displayed. ECHO ON entered interactively restores the display. If ECHO is entered interactively without a parameter, the current status of the ECHO flag is displayed.

ECHO can also be followed by a message to be sent to standard output regardless of the status of the ECHO flag (on or off). Note that if ECHO is on, two copies of the message are actually displayed, the first copy preceded by the word *ECHO*. ECHO *message* is frequently used to display prompts and informative text during the execution of a batch file because text following REM or PAUSE commands is not displayed if ECHO is off.

ECHO *message* can also be used to build lists or other batch files dynamically while the batch file is executing. For example, the messages in the following ECHO commands are used to build the file STARTUP.BAT:

```
ECHO CHKDSK > STARTUP.BAT
ECHO DIR /W >> STARTUP.BAT
ECHO PROMPT $p$g >> STARTUP.BAT
```

The first ECHO command causes the message *CHKDSK* to be redirected to the file STARTUP.BAT. The second and third ECHO commands cause the messages *DIR/W* and *PROMPT pg* to be appended to the existing contents of STARTUP.BAT. The completed STARTUP.BAT file contains the following:

```
CHKDSK
DIR /W
PROMPT $p$g
```

Note: When the pipe symbol (¦) is used in *message*, the symbol and any characters following it are ignored until a redirection symbol (<, >, or >>) is encountered, at which point the redirection symbol and the remaining characters are recognized. For example, if the line

```
ECHO DIR ¦ SORT > STARTUP.BAT
```

was placed in a batch file and subsequently executed, the only characters echoed to the file STARTUP.BAT would be *DIR*; the pipe symbol and the characters between it and the redirection symbol > would be ignored.

Examples

To disable the display of each batch-file command as it is executed, include the following line as the first line in the batch file:

```
ECHO OFF
```

To display the message *Now formatting disk* on standard output, include the following line in the batch file:

```
ECHO Now formatting disk
```

To display the current status of the ECHO flag, include the following line in the batch file:

```
ECHO
```

If the ECHO flag is currently off, MS-DOS displays:

```
ECHO is off
```

To echo a blank line to the screen with versions 2.x, type a space after the ECHO command and press Enter. To echo a blank line with versions 3.x, type the ECHO command and a space, then hold down Alt and type 255 on the numeric keypad; finally, release the Alt key and press Enter.

Messages

ECHO is off

or

ECHO is on

If the ECHO command is entered without a parameter, one of these lines is displayed to give the current status of the batch processor's ECHO flag.

BATCH: FOR

2.0 and later

Execute Command on File Set

Internal

Purpose

Executes a command or program for each file in a set of files.

Syntax

FOR %%*variable* IN (*set*) DO *command* (batch processing)

or

FOR %*variable* IN (*set*) DO *command* (interactive processing)

where:

variable	is a variable name that can be any single character except the numerals 0 through 9, the redirection symbols (<, >, and >>), and the pipe symbol (|); case is significant.
set	is one or more filenames, pathnames, character strings, or metacharacters, separated by spaces, commas, or semicolons; wildcard characters are permitted in filenames.
command	is any MS-DOS command or program except the FOR command; the variable name %%*variable* (or %*variable* in interactive mode) can be part of the command.

Description

The FOR command allows sequential execution of the same command or program on each member of a set of files.

The *set* parameter can contain multiple filenames (including wildcards), pathnames, character strings, or metacharacters such as the replaceable parameters %0 through %9. Each of the following lines is an example of a valid set:

```
(FILE1.TXT %1 %2 B:\PROG\LISTING?.TXT)
(A:\%1 A:\%2 C:\LETTERS\*.TXT C:MEMO?.*)
(%PATH%)
```

Each filename from *set* is assigned in turn to %*variable* and then the specified command or program is executed. (When the FOR command line is executed in a batch file, the leading percent sign of %%*variable* is removed, leaving %*variable*.) If a filename in *set* contains wildcards, each matching file is used before the batch processor goes on to the next member of *set*.

Note: In versions 2.x, *set* can consist only of a list of single filenames, a single filename with wildcard characters, or a combination of single filenames and metacharacters. In versions 3.x, however, all combinations of these are allowed in the same set.

The FOR command can also be used interactively at the MS-DOS prompt to perform a single command on several files without entering the same command for each file. When FOR is used in this manner, only one percent sign (%) should be used before the dummy alphabetic variable; in this case, the percent sign is not removed during processing. When the FOR command is used interactively, environment variables such as %PATH% cannot be used as part of the filename set.

Examples

To view all the files with the extension .TXT in the current directory, include the following line in the batch file:

```
FOR %%X IN (*.TXT) DO TYPE %%X
```

To perform the same function interactively, type

```
C>FOR %X IN (*.TXT) DO TYPE %X  <Enter>
```

To copy up to nine files to the disk in drive A, specifying the names of the files in the batch-file command line, include the following line in the batch file:

```
FOR %%Y IN (%1 %2 %3 %4 %5 %6 %7 %8 %9) DO COPY %%Y A:
```

(Recall that %0 is the name of the batch file.)

To execute successive batch files under the control of one batch file, use the /C switch with COMMAND, as in the following batch-file line:

```
FOR %%Z IN (BAT1 BAT2 BAT3) DO COMMAND /C %%Z
```

Message

FOR cannot be nested
The command or program performed by a FOR command cannot be another FOR command.

BATCH: GOTO

2.0 and later

Jump to Label

Internal

Purpose

Transfers program control to the batch-file line following the specified label.

Syntax

GOTO *name*

where:

name is a batch-file label declared elsewhere in the file in the form *:name*.

Description

The GOTO command causes the batch-file processor to transfer its point of execution to the line following the specified label. If the label does not exist in the file, execution of the batch file is terminated with the message *Label not found*.

A batch-file label is defined as a line with a colon character (:) in the first column, followed by any text (including spaces but not other separator characters such as semicolons or equal signs). Only the first eight characters following the colon are significant; spaces are not counted in the eight characters.

Examples

The GOTO command is frequently used in combination with the IF and SHIFT batch commands to perform some action based on the return code from a program. For example, the following batch file will back up a variable number of files or directories, whose names are specified in the batch-file command line, to a floppy disk in drive A. The batch file accomplishes this by executing the BACKUP program with successive pathnames specified in the command line until BACKUP returns a nonzero (error) code. Control is then transferred to the label *:DONE*, and the batch file is terminated.

```
1   ECHO OFF
2   :START
3   BACKUP %1 A:
4   IF ERRORLEVEL 1 GOTO DONE
5   SHIFT
6   GOTO START
7   :DONE
```

Note that the batch file includes two labels, *:START* and *:DONE*, in lines 2 and 7, respectively. It also includes two GOTO commands, in lines 4 and 6. (The line numbers in the listing above are included only for reference and are not present in the actual batch file.) If the condition in line 4 is true (the BACKUP program returned an exit code of 1 or higher), the remainder of line 4 is executed and program control passes to the *:DONE* label in

line 7. If the condition is false, program control passes to line 5, the SHIFT command is executed, and program control goes to line 6, where the GOTO statement returns program control to line 2.

Message

Label not found
The specified label does not exist in the batch file.

BATCH: IF

2.0 and later

Perform Conditional Execution

Internal

Purpose

Tests a condition and executes a command or program if the condition is met.

Syntax

IF [NOT] *condition command*

where:

condition is one of the following:

ERRORLEVEL *number*
The condition is true if the exit code of the program last executed by COMMAND.COM was equal to or greater than *number*. Note that not all MS-DOS commands return explicit exit codes.

string1==string2
The condition is true if *string1* and *string2* are identical after parameter substitution; case is significant. The strings cannot contain separator characters such as commas, semicolons, equal signs, or spaces.

EXIST *pathname*
The condition is true if the specified file exists. The pathname can include metacharacters.

command is the command or program to be executed if the condition is true.

Description

The IF command provides conditional execution of a command or program in a batch file. When *condition* is true, IF executes the specified command, which can be another IF command, any other MS-DOS internal command, or a program. When *condition* is not true, MS-DOS ignores *command* and proceeds to the next line in the batch file. The sense of any condition can be reversed by preceding the test or expression with NOT.

Examples

To branch to the label *:ERROR* if the file LEDGER.DAT does not exist, include the following line in the batch file:

```
IF NOT EXIST LEDGER.DAT GOTO ERROR
```

To branch to the label *:ONEPAR* if the batch-file command line does not contain at least two parameters, include the following line in the batch file:

```
IF "%2"==""GOTO ONEPAR
```

or

```
IF %2~==~ GOTO ONEPAR
```

Note that the existence of a replaceable parameter can be determined by concatenating it to another string. In the first example, quotation marks are concatenated on either side of the replaceable parameter; if %2 doesn't exist, *"%2"=="* evaluates to *""=="*, which is true and will allow *GOTO ONEPAR* to be executed. In the second example, a tilde character is concatenated to the end of the replaceable parameter; if %2 doesn't exist, the argument becomes ~==~.

To copy the file specified by the first replaceable batch-file parameter to drive A only if it does not already exist on the disk in drive A, include the following line in the batch file:

```
IF NOT EXIST A:%1 COPY %1 A:
```

To branch to the label *:DONE* if the first replaceable batch-file parameter exists in the \PROG directory on drive C *and* in the \BACKUP directory on drive C, include the following line in the batch file:

```
IF EXIST C:\PROG\%1 IF EXIST C:\BACKUP\%1 GOTO DONE
```

Messages

Bad command or filename
The command following the condition in the IF statement was misspelled, does not exist, or was represented by a replaceable parameter that was not supplied in the command line that invoked the batch file.

Syntax error
The condition specified in the IF statement cannot be tested.

BATCH: PAUSE

1.0 and later

Suspend Batch-File Execution

Internal

Purpose

Displays a message, suspends execution of a batch file, and waits for the user to press a key.

Syntax

PAUSE [*message*]

where:

message is a text string to be displayed on standard output.

Description

The PAUSE command displays the message *Strike a key when ready...* and suspends execution of a batch file until the user presses a key. This command can be used to allow time for the operator to change disks, change the type of forms on the printer, or take some other action that is necessary before the batch file can continue.

If the batch processor's ECHO flag is on when the PAUSE command is executed, the entire line containing the PAUSE statement is displayed on the screen so that the optional message is visible to the user. The message *Strike a key when ready...* is then displayed on a new line and the system waits. Note that *Strike a key when ready...* is *always* displayed, even if the ECHO flag is off. When the user presses a key, execution of the batch file resumes.

Note: Redirection symbols should not be used within *message*. They prevent the message *Strike a key when ready...* from being displayed on the screen.

If the user presses Ctrl-C or Ctrl-Break while a PAUSE command is waiting for a key to be pressed, a prompt is displayed that gives the user the opportunity to terminate the execution of the batch file. This same message is displayed whenever the user presses Ctrl-C or Ctrl-Break during the execution of a batch file; however, using PAUSE commands supplemented by appropriate ECHO commands at strategic points within a batch file provides the user with clearly defined breakpoints for terminating the file.

Examples

To display the message *Put an empty disk in drive A* and then wait until the user has pressed a key, include the following line in the batch file:

```
PAUSE Put an empty disk in drive A
```

When this line of the batch file is executed, if the ECHO flag is on, the user sees the following messages on the screen:

```
C>PAUSE Put an empty disk in drive A
Strike a key when ready . . .
```

If the ECHO flag is off, only the message *Strike a key when ready...* appears.

To display the message without the prompt and command, the PAUSE command can be used immediately after an ECHO command, as follows:

```
ECHO OFF
CLS
ECHO Put an empty disk in drive A
PAUSE
```

This batch file will display the following message on the screen:

```
Put an empty disk in drive A
Strike a key when ready . . .
```

Note that the message must be included in an ECHO command. With ECHO off, a PAUSE message is not displayed.

BATCH: REM

1.0 and later

Include Comment Line

Internal

Purpose

Designates a remark, or comment, line in a batch file.

Syntax

REM [*message*]

where:

message is any text.

Description

The REM command allows inclusion of remarks, or comments, within a batch file. Remarks are often used to document the purpose of other commands within the file for the benefit of those who may wish to modify the file later.

If the ECHO flag is on, remarks are displayed on the screen during the execution of a batch file. Thus, remarks can also be used to provide information, guidance, or prompts to the user; however, the ECHO and PAUSE commands are more suitable for these purposes.

REM can also be used alone to insert blank lines in a batch file to improve readability. (If ECHO is on, the word *REM* will still be displayed.)

Note: The redirection symbols (<, >, and >>) and piping character (¦) produce no meaningful results with the REM command and should not be used.

Example

To document a batch file's revision history with the internal comment *This batch file last modified on 6/18/87*, include the following line in the batch file:

```
REM This batch file last modified on 6/18/87
```

BATCH: SHIFT

Shift Replaceable Parameters

2.0 and later

Internal

Purpose

Changes the position of the replaceable parameters in a batch-file command line, thereby allowing more than 10 replaceable parameters.

Syntax

SHIFT

Description

Ordinarily only 10 replaceable parameters (%0 through %9, where %0 is the name of the batch file) can be referenced within a batch file. The SHIFT command allows access to additional parameters specified in the command line by shifting the contents of each of the previously assigned parameters to a lower number (%1 becomes %0, %2 becomes %1, and so on). The previous contents of %0 are lost and are not recoverable. The eleventh parameter in the batch-file command line is then moved into %9. This allows more than 10 parameters to be specified in the batch-file command line and subsequently processed in the batch file.

Example

The following batch file will copy a variable number of files, whose names are entered in the batch-file command line, to the disk in drive A:

```
ECHO OFF
:NEXT
IF "%1"=="" GOTO DONE
COPY %1 A:
SHIFT
GOTO NEXT
:DONE
```

BREAK

2.0 and later

Set Control-C Check

Internal

Purpose

Sets or clears MS-DOS's internal flag for Control-C checking.

Syntax

BREAK [ON|OFF]

Description

Pressing Ctrl-C or Ctrl-Break while a program is running ordinarily terminates the program, unless the program itself contains instructions that disable MS-DOS's Control-C handling. As a rule, MS-DOS checks the keyboard for a Control-C only when a character is read from or written to a character device (keyboard, screen, printer, or auxiliary port). Therefore, if a program executes for long periods without performing such character I/O, detection of the user's entry of a Control-C may be delayed. The BREAK ON command causes MS-DOS to also check the keyboard for a Control-C at the time of each system call (which slows the system somewhat); the BREAK OFF command disables such extended Control-C checking. The default setting for BREAK is off.

If the BREAK command is entered alone, the current status of MS-DOS's internal BREAK flag is displayed.

Examples

To display the current status of the MS-DOS internal flag for extended Control-C checking, type

```
C>BREAK  <Enter>
```

MS-DOS displays

```
BREAK is off
```

or

```
BREAK is on
```

depending on the status of the BREAK flag.

To enable extended checking for Control-C during disk operations, type

```
C>BREAK ON  <Enter>
```

Messages

BREAK is on

or

BREAK is off
Extended Control-C checking is enabled or disabled, respectively. These messages occur in response to a BREAK status check.

Must specify ON or OFF
An invalid parameter was supplied in a BREAK command.

CHDIR or CD

2.0 and later

Change Current Directory

Internal

Purpose

Changes the current directory or displays the current path of the specified or default disk drive.

Syntax

CHDIR [*drive*:][*path*]

or

CD [*drive*:][*path*]

where:

drive is the letter of the drive for which the current directory will be changed or displayed, followed by a colon. Note that use of the *drive* parameter does not change the currently active drive.

path is one or more directory names, separated by backslash characters (\), that define an existing path.

Description

The CHDIR command, when followed by an existing path, is used to set the working directory for the default or specified disk drive.

The *path* parameter consists of the name of an existing directory, optionally followed by the names of existing subdirectories, each separated from the next by a backslash character. If *path* begins with a backslash, CHDIR assumes that the first named directory is a subdirectory of the root directory; otherwise, CHDIR assumes that the first named directory is a subdirectory of the current directory. The special directory name .. , which is an alias for the parent directory of the current directory, can be used as the path.

When CHDIR is entered alone or with only a drive letter followed by a colon, the full path of the current directory for the default or specified drive is displayed.

CD is simply an alias for CHDIR; the two commands are identical.

Examples

To change the current directory for the current (default) disk drive to the path \V2\SOURCE, type

```
C>CD \V2\SOURCE  <Enter>
```

To display the name of the current directory for the disk in drive D, type

```
C>CD D:  <Enter>
```

To return to the parent directory of the current directory, type

```
C>CD ..  <Enter>
```

Messages

Invalid directory
One of the directories in the specified path does not exist.

Invalid drive specification
An invalid drive letter was given or the named drive does not exist in the system.

CHKDSK

1.0 and later

Check Disk Status

External

Purpose

Analyzes the allocation of storage space on a disk and displays a summary report of the space occupied by files and directories.

Syntax

CHKDSK [*drive*:][*pathname*] [/F] [/V]

where:

drive	is the letter of the drive containing the disk to be analyzed, followed by a colon.
pathname	is the location and, optionally, the name of the file(s) to be checked for fragmentation; wildcard characters are permitted in the filename.
/F	repairs errors (versions 2.0 and later).
/V	"verbose mode," reports the name of each file as it is checked (versions 2.0 and later).

Description

The CHKDSK command analyzes the disk directory and file allocation table for consistency and reports any errors. If the /V switch is included in the command line, the name of each file processed is displayed as the disk is being analyzed.

After analyzing the disk, CHKDSK displays a summary of the disk and RAM space used and available. The disk-space report includes

- Total disk space in bytes
- Number of bytes allocated to hidden files
- Number of bytes contained in directories
- Number of bytes contained in user files
- Number of bytes contained in bad (unusable) sectors
- Number of available bytes on the disk

(Hidden files are files that do not appear in a directory listing. A bootable MS-DOS or PC-DOS disk always contains two hidden files—MSDOS.SYS and IO.SYS or IBMDOS.COM and IBMBIO.COM, respectively—that contain the operating system. A volume label, if present, counts as a hidden file. In addition, some application programs create hidden files for copy protection or other purposes.)

Directory errors detected by CHKDSK include

- Invalid pointers to data areas
- Bad file attributes in directory entries

- Damage to a portion of the directory that makes it impossible to check one or more paths
- Damage to an entire directory that makes the files contained in that directory inaccessible

File allocation table (FAT) errors detected by CHKDSK include

- Defective disk sectors in the FAT
- Invalid cluster (disk allocation unit) numbers in the FAT
- Lost clusters
- Cross-linking of files on the same cluster

If the /F switch is included in the command line, CHKDSK will attempt to repair errors in disk allocation and recover as much data as possible. Because repairs usually involve changes to the disk's file allocation table that may cause a loss of information, the user is prompted for confirmation. Lost clusters are collected into files in the root directory with names of the form FILE*nnnn*.CHK.

If the command line contains a file specification, CHKDSK will examine all files that match the specification and report on their fragmentation—that is, on whether or not their sectors are contiguous on the disk. (Fragmented files can degrade the performance of the system because of the time required to move the drive head back and forth across the disk to reach the various parts of the file.) Files on a floppy disk can be collected into contiguous sectors by copying them to an empty floppy disk. Files on a fixed disk can be collected into contiguous sectors by backing them all up to floppy disks, erasing all files and subdirectories on the fixed disk, and then restoring the files from the floppy disk.

Warning: CHKDSK should not be used on a network drive or on a drive created or affected by an ASSIGN, JOIN, or SUBST command.

Examples

To check the disk in the current drive, type

```
C>CHKDSK  <Enter>
```

If CHKDSK finds no errors, a report such as the following is displayed:

```
Volume HARDDISK    created Jun 8, 1986 9:34a

 21204992 bytes total disk space
    38912 bytes in 3 hidden files
   116736 bytes in 53 directories
 17055744 bytes in 715 user files
    20480 bytes in bad sectors
  3973120 bytes available on disk

   655360 bytes total memory
   566576 bytes free
```

Note that the line containing the volume name and creation date does not appear if the disk has not been assigned a volume name.

If CHKDSK finds errors, a message such as the following is displayed:

```
Errors found, F parameter not specified.
Corrections will not be written to disk.

10 lost clusters found in 3 chains.
Convert lost chains to files  (Y/N)?
```

A *Y* response at this point does not convert the lost chains to files; to do this, enter the CHKDSK command again with the /F switch specified.

To correct any allocation errors found by the CHKDSK command, type

```
C>CHKDSK /F  <Enter>
```

In this example, CHKDSK displays its usual report, followed by an error message:

```
Volume HARDDISK    created Jun 8, 1986 9:34a

 21204992 bytes total disk space
    38912 bytes in 3 hidden files
   116736 bytes in 53 directories
 17055744 bytes in 715 user files
    20480 bytes in bad sectors
  3973120 bytes available on disk

   655360 bytes total memory
   566576 bytes free

10 lost clusters found in 3 chains.
Convert lost chains to files (Y/N) ?
```

A *Y* response causes CHKDSK to recover the lost chains of clusters into files in the root directory, giving the files the names FILE0000.CHK, FILE0001.CHK, FILE0002.CHK, and so on. An *N* response causes CHKDSK to free the lost chains of clusters without saving the contents to files.

To check all files in the directory C:\SYSTEM with the extension .COM for fragmentation, type

```
C>CHKDSK C:\SYSTEM\*.COM  <Enter>
```

CHKDSK displays its usual report, followed by a list of fragmented files:

```
Volume HARDDISK    created Jun 8, 1986 9:34a

 21204992 bytes total disk space
    38912 bytes in 3 hidden files
   116736 bytes in 53 directories
 17055744 bytes in 715 user files
    20480 bytes in bad sectors
  3973120 bytes available on disk

   655360 bytes total memory
   566576 bytes free

C:\SYSTEM\ALUSQ.COM
   Contains 2 non-contiguous blocks.

C:\SYSTEM\EJECT.COM
   Contains 4 non-contiguous blocks.
```

Messages

. Does not exist.

or

.. Does not exist.
The . (alias for the current directory) or the .. (alias for the parent directory) entry is missing.

filename* Is cross linked on cluster *n
Two or more files have been assigned the same cluster. Make a copy of both files on another disk and then delete them from the disk containing the error. One or both of the resulting files may contain information belonging to the other file.

***x* lost clusters found in *y* chains.**
Convert lost chains to files (Y/N)?
Clusters have been identified that are not assigned to any existing file. If the /F switch was included in the original command line, respond with *Y* to convert the lost clusters to files in the root directory of the disk with names of the form FILE*nnnn*.CHK. If desired, the recovered clusters can then be returned to the free-disk-space pool by erasing the .CHK files.

Allocation error, size adjusted.
The size of the file indicated in the disk directory is not consistent with the number of clusters allocated to the file. If the /F switch was included in the command line, the file is truncated to the size indicated in the disk directory.

All specified file(s) are contiguous.
The clusters belonging to the specified file(s) are allocated contiguously (without fragmentation).

Cannot CHDIR to *pathname*
tree past this point not processed.
The tree directory structure of the disk being checked cannot be traveled to the specified directory. This message indicates severe damage to the disk's directories or files.

Cannot CHDIR to root
Processing cannot continue.
In traversing the tree directory structure of the disk being checked, CHKDSK was unable to return to the root directory. This message indicates severe damage to the disk's directories or files.

Cannot CHKDSK a Network drive
The drive containing the disk to be checked has been assigned to a network.

Cannot CHKDSK a SUBSTed or ASSIGNed drive
The drive containing the disk to be checked has been substituted or assigned.

Cannot recover . entry, processing continued.
The special directory entry **.** (alias for the current directory) is defective.

Cannot recover .. entry,
Entry has a bad attribute

or

Cannot recover .. entry,
Entry has a bad link

or

Cannot recover .. entry,
Entry has a bad size
The special directory entry **..** (alias for the parent directory of the current directory) is defective due to a bad attribute, link, or size.

CHDIR .. failed, trying alternate method.
While checking the tree structure, CHKDSK was unable to return to the parent directory of the current directory. It will attempt to return to that directory by starting over at the root directory and searching again.

Contains *n* non-contiguous blocks.
The clusters assigned to the specified file are not allocated contiguously on the disk.

Directory is joined
CHKDSK cannot process directories that have been joined using the JOIN command. Use the JOIN /D command to unjoin the directories, then run CHKDSK again.

Directory is totally empty, no . or ..
The specified directory does not contain the usual aliases for the current and parent directories. This message indicates severe damage to the disk's directories or files. Delete the directory and recreate it.

Disk error reading FAT *n*

or

Disk error writing FAT *n*
One of the file allocation tables for the disk being checked contains a defective sector. MS-DOS will use the alternate FAT if one is available. It is advisable to copy all the files on the disk containing the defective sector to another disk.

Errors found, F parameter not specified.
Corrections will not be written to disk.
Errors were found on the disk being checked, but the /F switch was not included in the command line.

File allocation table bad drive *X*:
The disk is not an MS-DOS disk. Repeat CHKDSK with the /F option; if this message is displayed again, reformat the disk.

File not found.
CHKDSK was unable to find the specified file.

First cluster number is invalid, entry truncated.
The directory entry for the specified file contains an invalid pointer to the disk's data area. If the /F switch was included in the command line, the file is truncated to a zero-length file.

General Failure error reading drive *X*:
The format of the disk being checked is not compatible with MS-DOS or the disk has not been formatted for use by MS-DOS.

Has invalid cluster, file truncated.
The file directory contains an invalid pointer to the disk's data area. If the /F switch was included in the command line, the file is truncated to a zero-length file.

Incorrect DOS version
The version of CHKDSK is not compatible with the version of MS-DOS that is running.

Insufficient memory
Processing cannot continue.
The computer does not have enough memory to contain the tables necessary for CHKDSK to process the specified disk.

Insufficient room in root directory.
Erase files in root and repeat CHKDSK.
The root directory is full and does not have room for the entries for recovered files. Delete some files from the root directory of the disk being checked and rerun the CHKDSK program.

Invalid current directory
Processing cannot continue.
The directory structure of the disk is so badly damaged that the disk is unusable.

Invalid drive specification
The CHKDSK command contained an invalid disk drive.

Invalid parameter
One of the switches in the command line is invalid.

Invalid sub-directory entry.
The directory name specified in the command line does not exist or is invalid.

Path not found.
One of the directories in the path specified in the command line does not exist or is invalid.

Probable non-DOS disk
Continue (Y/N) ?
The disk being checked was not formatted by MS-DOS or the file allocation table has been severely damaged or destroyed.

Unrecoverable error in directory.
Convert directory to file (Y/N)?
The specified directory is damaged and unusable. If the /F switch was included in the original command line, respond with *Y* to convert the damaged directory to a file in the root directory of the disk with a name of the form FILE*nnnn*.CHK. If desired, the .CHK file can then be deleted. Any files that were previously reached through the damaged directory will be lost.

CLS

Clear Screen

2.0 and later

Internal

Purpose

Clears the video display.

Syntax

CLS

Description

The CLS command clears the video display and displays the current prompt.

In some implementations of MS-DOS, proper operation of the CLS command may require installation of the ANSI.SYS console driver with a *DEVICE=ANSI.SYS* command in the CONFIG.SYS file.

Examples

To clear the screen, type

```
C>CLS  <Enter>
```

To save the ANSI escape sequence used by the CLS command (ESC[2J) into a file named CLEAR.TXT, type

```
C>CLS > CLEAR.TXT  <Enter>
```

COMMAND

1.0 and later

Command Processor

External

Purpose

Loads a secondary copy of the MS-DOS default command processor.

Syntax

COMMAND [*drive*:][*path*] [*device*] [/E:*n*] [/P] [/C *string*]

where:

path	is the name of the directory to be searched for COMMAND.COM when the transient portion needs to be reloaded; a drive letter can be included with versions 2.0 and later.
device	is the name of a character device to be used instead of CON for the command processor's input and output (versions 2.0 and later).
/E:*n*	is the initial size, in bytes, of the command processor's environment block (160–32768, default = 160) (version 3.2).
/P	fixes the newly loaded command processor permanently in memory (versions 2.0 and later).
/C *string*	causes the command processor to behave as a transient program and execute the command or program specified by *string* (versions 2.0 and later).

Description

The command processor is the module of the operating system that is responsible for issuing prompts to the user, interpreting commands, loading and executing transient application programs, and interpreting batch files. The file COMMAND.COM contains the MS-DOS default command processor, or shell. It is ordinarily loaded from the root directory of the system disk when the system is turned on or restarted, unless the SHELL command is used in the CONFIG.SYS file to specify another command processor or an alternate location for COMMAND.COM.

With versions 1.x, COMMAND.COM is invoked by the COMMAND command in response to a shell prompt or within a batch file. A second copy of the resident portion of COMMAND.COM is loaded and the memory occupied by the original resident portion is lost. The second copy of the transient portion simply overlays the original transient portion. (Versions 1.x of COMMAND support no switches or other parameters and any specified in the command line are ignored.) With versions 2.0 and later, the new copy of COMMAND.COM is loaded *in addition to* the parent command processor and serves as a secondary command processor.

The *path* parameter specifies the location of the COMMAND.COM file that is used to reload the transient part of the command processor if it is overlaid by application programs. If absent, *path* defaults to the root directory of the system (startup) disk.

The *device* parameter allows a character device other than CON to be used by the command processor for input and output. For example, use of AUX as the *device* parameter allows a personal computer to be controlled from a terminal attached to a serial port, instead of from the usual built-in keyboard and memory-mapped video display.

The secondary copy of COMMAND.COM ordinarily remains in memory and serves as the active command processor until an EXIT command is entered. If a /P switch is used with the COMMAND command, the new copy of COMMAND.COM is fixed in memory and the EXIT command is disabled. In such cases, the memory occupied by previously loaded copies of COMMAND.COM is simply lost.

The /E:*n* switch controls the size of the environment block initially allocated for the command processor. The default size of the block is 160 bytes, but the /E:*n* switch allows the initial allocation to be as large as 32768 bytes. This switch is frequently used when *COMMAND.COM* is included in the SHELL command in the CONFIG.SYS file.

When the /C *string* switch is included in the command line, followed by a string designating a command or program name, the new copy of COMMAND.COM carries out the operation specified by *string* and then exits, returning control to its parent command processor or other program. This option allows a batch file to invoke another batch file and then resume its own execution. (If a batch file names another batch file directly without using COMMAND /C *string* as an intermediary, the first batch file is terminated.) Note that when the /C *string* switch is used in combination with other switches, it must be the last switch in the command line.

A secondary copy of COMMAND.COM always inherits a copy of the environment of the command processor or other program that loaded it. Changes made to the new COMMAND.COM's environment with a SET, PROMPT, or PATH command do not affect the environment of any previously loaded program or command processor.

Examples

To execute the batch file MENU2.BAT from the batch file MENU1.BAT and then resume execution of MENU1.BAT, include the following line in MENU1.BAT:

```
COMMAND /C MENU2
```

To cause COMMAND.COM to be loaded from the directory \SYSTEM on drive C rather than from the root directory and to allocate an initial environment block of 1024 bytes, include the following line in the CONFIG.SYS file:

```
SHELL=C:\SYSTEM\COMMAND.COM C:\SYSTEM /P /E:1024
```

Messages

Bad or missing command interpreter
The file COMMAND.COM is not present in the root directory of the system disk and no SHELL command is present to specify an alternate command processor file or location, or the location specified for COMMAND.COM in a SHELL command is not correct. This message may also be seen if COMMAND.COM is moved from its original location after the system is booted.

Invalid device
The character device specified in the command line is not valid or does not exist.

Invalid environment size specified
The value supplied with the /E:*n* switch was less than 160 bytes or greater than 32768 bytes.

COMP

IBM

Compare Files

External

Purpose

Compares two files or sets of files. This command is available only with PC-DOS.

Syntax

COMP [*primary*] [*secondary*]

where:

primary	is the name of the file to be compared against and can be preceded by a drive and/or path; wildcard characters are permitted in the filename.
secondary	is the name of the file to be compared with *primary* and can be preceded by a drive and/or path; wildcard characters are permitted in the filename.

Description

The COMP command compares one file or set of files with another. As each pair of files is compared, the program reports whether the files are identical, different in size, or the same size but different in content.

The *primary* and *secondary* parameters can be any combination of drive, path, and filename, optionally including wildcards to allow sets of files to be compared. (With versions 1.x, using wildcards does not cause multiple file comparisons — only the first secondary file whose name matches the first primary filename is compared.) The *primary* parameter generally designates the specific files to be compared; the *secondary* parameter is usually only a drive and/or path, except when the files being compared have different names or extensions.

If both *primary* and *secondary* are omitted from the command line, the COMP program prompts for them interactively. If *primary* is given as a drive or path only, COMP assumes *.* to be the primary file. If *secondary* is given as a drive or path only, COMP compares all files on that drive or path whose filenames match those of the primary files.

The COMP command is included only with PC-DOS. MS-DOS versions 2.0 and later provide a similar function in the FC command, which also displays the differences between files.

Examples

To compare the file MYFILE.DAT on the disk in drive A with the file LEDGER.DAT on the disk in drive B, type

```
C>COMP A:MYFILE.DAT B:LEDGER.DAT  <Enter>
```

To compare all the files in the current directory of the disk in drive A with the corresponding files in the current directory of the disk in drive D, type

```
C>COMP A:*.* D:  <Enter>
```

To compare all the files with the extension .ASM in the directory C:\SOURCE with the corresponding files with extension .BAK on the disk in drive B, type

```
C>COMP C:\SOURCE\*.ASM B:*.BAK  <Enter>
```

Messages

10 mismatches - ending compare
The primary and secondary files are the same size but have more than 10 internal differences. The compare operation on this pair of files is aborted and COMP proceeds to the next pair of files, if any.

filename* and *filename
This informational message shows the full filenames of the two files currently being compared.

Access Denied
An attempt was made to compare a locked file.

Cannot compare file to itself
An attempt was made to compare a file with itself.

Compare error at OFFSET *nn*
File 1 = *nn*
File 2 = *nn*
This informational message itemizes the first 10 differences in data between the two files being compared (if the files are the same size), displaying the file offset and the differing bytes from each file as hexadecimal values.

Compare more files (Y/N)?
After all specified pairs of files have been compared, the COMP program allows the entry of another pair of file specifications. Respond with *Y* or press Enter to continue; respond with *N* to terminate the COMP program.

Enter 2nd file name or drive id
If the secondary filename was not specified in the COMP command, this message prompts the user to enter it (or a path, if the secondary file has the same name as the primary file).

Enter primary file name
If no parameter was entered after COMP, this message prompts the user to enter the primary filename. If a drive or path is specified, COMP assumes *.* for the primary filename.

EOF mark not found
The last byte at the logical end of the file was not a Control-Z character (^Z, or 1AH). This message is commonly seen during comparison of two files that are not ASCII text files, such as executable program files.

Files compare OK
The files being compared were the same length and contained identical data.

File not found
The specified filename was invalid or the file does not exist.

Files are different sizes
The two files being compared have different sizes recorded in the directory. No comparison on the data within the files is attempted.

File sharing conflict
COMP is unable to compare the two current files because one of the files is in use by another process.

Incorrect DOS version
The version of COMP is not compatible with the version of PC-DOS that is running.

Insufficient memory
The available system memory is insufficient to run the COMP program.

Invalid drive specification
The drive specification in *primary* or *secondary* is invalid or does not exist.

Invalid path
The path or directory in *primary* or *secondary* is invalid or does not exist.

Too many files open
No more system file handles are available. Increase the value of the FILES command in the CONFIG.SYS file and restart the system.

CONFIG.SYS

2.0 and later

System Configuration File

Purpose

Allows the user to configure the operating system.

Description

The CONFIG.SYS file is an ASCII text file that MS-DOS processes during initialization (when the system is turned on or restarted). It allows the user to configure certain aspects of the operating system, such as the number of internal disk buffers allocated, the number of files that can be open at one time, the formats for date and currency, and the name and location of the executable file containing the command processor. CONFIG.SYS can also contain commands that extend the system with installable device drivers for terminal emulation, virtual disks or RAMdisks, extended or expanded memory, and other special peripheral devices.

The CONFIG.SYS file can be created or modified with EDLIN or with any other editor or word processor that can produce ordinary ASCII text files (nondocument files) and save them to disk. The CONFIG.SYS file must be in the root directory of the disk that is used to start the operating system in order for it to be processed during system initialization. When changes are made to the CONFIG.SYS file, they do not take effect until the system is restarted.

Commands in the CONFIG.SYS file take the form

command[=]*value*

(Note that the equal sign is optional; any other valid MS-DOS separator [semicolon, tab, or space] can be used instead.) The commands supported are

Command	Action
BREAK	Controls extended checking for Control-C.
BUFFERS	Specifies the number of internal disk-sector buffers available for use by MS-DOS when reading from or writing to a disk.
COUNTRY	Controls date, time, and currency formatting.
DEVICE	Specifies the filename of an installable device driver.
DRIVPARM	Redefines the default characteristics of the resident MS-DOS block device(s) (version 3.2).
FCBS	Specifies the maximum number of simultaneously open file control blocks (versions 3.0 and later).

(more)

Command	Action
FILES	Specifies the maximum number of simultaneously open files controlled by handles.
LASTDRIVE	Sets the highest valid drive letter (versions 3.0 and later).
SHELL	Specifies the filename (and optionally the drive and/or path) of the system command processor.
STACKS	Sets the number and size of stack frames for the system.

Each of these commands is discussed in detail on the following pages.

Message

Unrecognized command in CONFIG.SYS
A command in the CONFIG.SYS file was misspelled, an invalid parameter was used, or a command was included that is not compatible with the version of MS-DOS that is running. Correct the CONFIG.SYS file and restart the system.

CONFIG.SYS: BREAK

2.0 and later

Configure Control-C Checking

Purpose

Sets or clears MS-DOS's internal flag for Control-C checking.

Syntax

BREAK=ON ¦OFF

Description

Pressing Ctrl-C or Ctrl-Break while a program is running ordinarily terminates the program, unless the program itself contains instructions that disable MS-DOS's Control-C handling. As a rule, MS-DOS checks the keyboard for a Control-C only when a character is read from or written to a character device (keyboard, screen, printer, or auxiliary port). Therefore, if a program executes for long periods without performing such character I/O, detection of the user's entry of a Control-C may be delayed. The BREAK=ON command causes MS-DOS to also check the keyboard for a Control-C at the time of each system call (which slows the system somewhat); the BREAK=OFF command disables such extended Control-C checking. The default setting for BREAK is off.

Extended Control-C checking can also be enabled or disabled at the command prompt with the interactive form of the BREAK command whenever the system is running.

Example

To enable extended Control-C checking during MS-DOS disk operations, insert the line

```
BREAK=ON
```

into the CONFIG.SYS file and restart the system.

Message

Unrecognized command in CONFIG.SYS
The setting supplied for the BREAK command was not ON or OFF. Correct the CONFIG.SYS file and restart the system.

CONFIG.SYS: BUFFERS

2.0 and later

Configure Internal Disk Buffers

Purpose

Sets the number of MS-DOS's internal disk buffers.

Syntax

BUFFERS=*nn*

where:

nn is the number of buffers (1–99, default = 2; default = 3 for IBM PC/AT and compatibles).

Description

MS-DOS maintains a set of internal buffers (sometimes referred to as a disk cache) in which it keeps copies of the sectors most recently read from or written to the disk. Whenever a program requests a disk read, MS-DOS first searches the disk buffers to determine whether a copy of the disk sector containing the required data is already present in RAM. If the sector is found, the actual disk access is bypassed. This technique can significantly improve the overall performance of the disk operating system.

By using the BUFFERS command in the CONFIG.SYS file, the user can control the number of buffers in MS-DOS's disk cache. The default number of buffers is 2 for an IBM PC, PC/XT, or compatible and 3 for an IBM PC/AT or compatible. The optimum number of buffers varies, depending in part on the characteristics and types of the system disk drives, the types of application programs used on the system, the number and levels of subdirectories in the file structure, and the amount of RAM in the system.

If the system has only floppy-disk drives, the default setting of 2 buffers is sufficient. If the system includes a fixed disk, increasing the number of buffers to 10 or so typically speeds up overall system operation. Configuring the system for too many buffers, however, can actually degrade the performance of the system.

Increases in the number of buffers should be tailored to the type of application most frequently used. For example, allocation of extra disk buffers will not improve the performance of programs that use primarily sequential file access but may considerably enhance the execution times of programs that perform random access on a relatively small number of disk records (such as the index for a database file). In addition, if the system has many subdirectories organized in several levels, increasing the number of buffers can significantly increase the speed of disk operations.

The ideal number of buffers for a given system is difficult to predict because of the interactions between the access time of the disk, the speed of the central processing unit, and the

RAM requirements and disk access behavior of the mix of application programs. However, a reasonably optimal number of buffers can be quickly estimated experimentally by increasing the number of buffers in increments of five or so, restarting the system, performing some simple timing tests on the most frequently used application programs, and observing at what number of buffers system performance begins to degrade.

Example

To allocate 20 internal disk buffers, insert the line

```
BUFFERS=20
```

into the CONFIG.SYS file and restart the system.

Message

Unrecognized command in CONFIG.SYS
The value supplied for the BUFFERS command was not a number in the range 1 through 99.

CONFIG.SYS: COUNTRY

2.1 and later

Set Country Code

Purpose

Configures MS-DOS's internationalization support for a specific country.

Syntax

COUNTRY=*nnn*

where:

nnn is the international telephone dialing prefix for the country (001–999, default = 001):

Australia	061
Belgium	032
Denmark	045
Finland	358
France	033
Israel	972
Italy	039
Netherlands	031
Norway	047
Spain	034
Sweden	046
Switzerland	041
United Kingdom	044
USA	001
West Germany	049

Note: In versions 2.x (except 2.0), *nnn* is 01 through 99. Individual computer manufacturers determine the specific codes supported by their versions of MS-DOS.

Description

The COUNTRY command enables the user to tailor MS-DOS's date, time, and currency displays for a specific country. This capability, termed internationalization support, is achieved through use of a country code that controls the contents of the table MS-DOS uses to format these displays (including numeric separators). (The internationalization table is made available to application programs through Interrupt 21H Function 38H.) Beginning with version 3.0, PC-DOS also supports the COUNTRY command.

Example

In West Germany, the format for the date is *dd.mm.yy*. To configure MS-DOS to use this date format, insert the line

```
COUNTRY=049
```

into the CONFIG.SYS file and restart the system.

Message

Invalid country code
The specified country code is not supported by the version of MS-DOS that is running.

CONFIG.SYS: DEVICE

2.0 and later

Install Device Driver

Purpose

Loads and links an installable device driver into the operating system during initialization.

Syntax

DEVICE=[*drive*:][*path*]*filename* [*options*]

where:

filename is the name of the device-driver file, optionally preceded by a drive and/or path.

options specifies any switches or other parameters needed by the device driver; the DEVICE command itself has no switches.

Description

Device drivers are the modules of the operating system that control the interface between the operating system and peripheral devices such as disk drives, magnetic-tape drives, CRT terminals, and printers.

As supplied, MS-DOS already contains device drivers for the keyboard, video display, serial port, printer, real-time clock, and disk devices. Device drivers for additional peripheral devices can be linked into the operating system by adding a DEVICE command to the CONFIG.SYS file, placing the file containing the device driver on the system startup disk (or at the location specified by the *drive*: and/or *path* parameter), and restarting the computer.

If a drive other than the one containing the system disk is named as the location of the device driver, that drive must either be accessible via the system's default disk driver or be a drive configured with a previous DEVICE command.

Most OEM implementations of version 3.2 provide three installable device drivers: ANSI.SYS, which allows the video display and keyboard to be controlled by ANSI standard escape sequences; DRIVER.SYS, which supports external disk drives; and RAMDRIVE.SYS (VDISK.SYS with PC-DOS), which uses a portion of the machine's RAM to emulate a disk drive. *See* USER COMMANDS: ANSI.SYS; DRIVER.SYS; RAMDRIVE.SYS; VDISK.SYS.

Many manufacturers of add-on products for MS-DOS machines (such as network interfaces or Lotus/Intel/Microsoft Expanded Memory boards) also supply installable device drivers for use with their hardware. For information concerning these drivers, see the product manufacturer's user's manual.

Examples

To load the ANSI standard console driver, insert the line

```
DEVICE=ANSI.SYS
```

into the CONFIG.SYS file, place the file ANSI.SYS in the root directory of the system disk, and restart the system.

To load the RAMDRIVE.SYS driver located in the \DRIVERS directory on the disk in drive A, configuring it for 1024 KB in extended memory, insert the line

```
DEVICE=A:\DRIVERS\RAMDRIVE.SYS 1024 /E
```

into the CONFIG.SYS file and restart the system.

Messages

Bad or missing *filename*
The filename specified in the DEVICE command is invalid or does not exist or the file does not contain a valid MS-DOS installable device driver.

Sector size too large in file *filename*
The specified installable device driver uses a sector size that is larger than the sector size used by any of the system's default disk drivers. Such a driver cannot be used because MS-DOS's internal disk buffers will not be large enough to hold a sector read from the device.

CONFIG.SYS: DRIVPARM

3.2

Set Block-Device Parameters

Purpose

Alters the system's list of characteristics for an existing block device.

Syntax

DRIVPARM=/D:*n* [/C] [/F:*n*] [/H:*n*] [/N] [/S:*n*] [/T:*n*]

where:

/D:*n* is the drive number (0–255; 0 = A, 1 = B, etc.) and must always be the first switch in the command line.

/C indicates that the device provides door-lock-status support.

/F:*n* is a form-factor index from the following table (default = 2 if the DRIVPARM command is present but this switch is omitted):

0	320 KB or 360 KB
1	1.2 MB
2	720 KB
3	8-inch single-density floppy disk
4	8-inch double-density floppy disk
5	Fixed disk
6	Tape drive
7	Other

/H:*n* is the number of read/write heads (1–99).

/N indicates that the block device is not removable.

/S:*n* is the number of sectors per track (1–99).

/T:*n* is the number of tracks per side (1–999).

Note: The DRIVPARM command must not be used to specify device characteristics that the device driver is not capable of supporting.

Description

Whenever the device driver for a block device such as a disk drive or magnetic-tape drive performs input or output, it refers to an internal table of characteristics for the device that allows it to convert logical addresses to physical addresses. The DRIVPARM command modifies the default MS-DOS values in the table of characteristics for a particular block device during system initialization (when the computer is turned on or restarted). Multiple DRIVPARM commands, each modifying the characteristics of a different block device, can be included in the same CONFIG.SYS file. Any characteristics not specifically altered in

the DRIVPARM command for a particular device retain their original values, except for /F:*n*, which defaults to 2.

DRIVPARM commands that alter the characteristics for block devices controlled by *installable* device drivers must follow the DEVICE command that loads the device driver itself.

Example

Assume that drive B is a floppy-disk drive originally configured for 40 tracks with 8 sectors per track. To reconfigure the drive to read or write 80 tracks of 9 sectors each, insert the line

```
DRIVPARM=/D:1 /S:9 /T:80
```

into the CONFIG.SYS file and restart the system. For this command to be valid the drive must be capable of supporting these parameters.

Message

Unrecognized command in CONFIG.SYS
An invalid parameter was specified in a DRIVPARM command.

CONFIG.SYS: FCBS

3.0 and later

Set Maximum Open Files Using File Control Blocks (FCBs)

Purpose

Configures the maximum number of files that can be open concurrently using file control blocks (FCBs). This command has no practical effect unless either the file-sharing support module SHARE.EXE or networking support has been loaded.

Syntax

FCBS=*m*,*p*

where:

m is the maximum number of files that can be open concurrently using FCBs (1–255, default = 4).

p is the number of files opened with FCBs that are protected against automatic closure (0–*m*, default = 0).

Description

MS-DOS supports two methods of file access: file control blocks and file handles. A file control block is a data structure that stores information about an open file. It resides inside an application program's memory space and is accessed by both MS-DOS and the application. (*See* USER COMMANDS: CONFIG.SYS: FILES for information on file handles.)

In a network environment, a large number of active FCBs or improper use of FCBs by an application can seriously degrade the performance of the network as a whole. Consequently, MS-DOS versions 3.0 and later provide the FCBS command to enable the user to limit the number of files that can be open concurrently using FCBs if either the file-sharing support module SHARE.EXE (*see* USER COMMANDS: SHARE) or network support has been loaded. If an application program attempts to exceed the specified number of files, MS-DOS closes the file with the least recently used FCB.

The *p* parameter in the FCBS command line allows the user to protect files from unilateral closure by MS-DOS. The value of *p* is the number of files, counting from the first file opened using an FCB, that cannot be closed automatically.

If the current value of FCBS is *4,0* (the default) when the file-sharing module SHARE.EXE or network support is loaded, MS-DOS automatically increases the maximum number of files that can be open concurrently to 16 and the number of files protected against automatic closure to 8. (When multiple FCBs refer to the same file, the file is counted only once.)

Examples

To set the maximum number of files that can be concurrently open using FCBs to 10 and protect none of the FCB-opened files against automatic closure by MS-DOS, insert the line

```
FCBS=10,0
```

into the CONFIG.SYS file and restart the system.

To set the maximum number of files that can be concurrently open using FCBs to 8 but protect the first 4 FCB-opened files against automatic closure by MS-DOS, insert the line

```
FCBS=8,4
```

into the CONFIG.SYS file and restart the system.

Message

Unrecognized command in CONFIG.SYS
An invalid number was specified as one of the parameters in the FCBS command.

CONFIG.SYS: FILES

2.0 and later

Set Maximum Open Files Using Handles

Purpose

Configures the maximum number of files and/or devices that can be open concurrently using file handles.

Syntax

FILES=*n*

where:

n is the maximum number of files and devices that can be open concurrently using file handles (8–255, default = 8).

Description

MS-DOS supports two methods of file access: file handles and file control blocks (FCBs). During initialization, MS-DOS allocates a data structure that holds information about files and/or devices opened with the handle, or extended-file-management, function calls. This structure resides inside the operating system's memory space and is accessed only by MS-DOS. (*See* USER COMMANDS: CONFIG.SYS: FCBS.) The default size of this data structure allows 8 files and/or devices to be open concurrently using the file-handle functions. The FILES command enables the user to change the size of the data structure. (Note that increasing the size of the data structure decreases the amount of RAM available to application programs.)

The FILES command controls the maximum number of files and/or devices opened with handles for *all* active processes in the system combined. The limit on the number of files and/or devices opened for a single process using handles is 20 or the number of entries in the allocated data structure, whichever is less. Five of the 20 possible handles for a given process are automatically assigned to standard input, standard output, standard error, standard auxiliary, and standard list. However, since standard input, standard output, and standard error all default to the same device (CON), only three of the allocated data-structure entries are actually expended. In addition, the preassigned standard device handles for a process can be closed and reused for other files and devices, if necessary.

Example

To set the maximum number of files and/or devices that can be concurrently open using the handle functions to 20, insert the line

```
FILES=20
```

into the CONFIG.SYS file and restart the system.

Message

Unrecognized command in CONFIG.SYS
An invalid number was specified in the FILES command.

CONFIG.SYS: LASTDRIVE

3.0 and later

Set Highest Logical Drive

Purpose

Defines the highest letter that MS-DOS will recognize as a disk-drive code.

Syntax

LASTDRIVE=*drive*

where:

drive is a single letter (A–Z).

Description

MS-DOS block devices (floppy-disk drives, fixed-disk drives, and magnetic-tape drives) are referred to by logical drive codes consisting of a single letter from A through Z. In most MS-DOS systems, drives A and B are floppy-disk drives, drive C is a fixed disk, and drives D and above are such devices as additional fixed disks, RAMdisks, or network volumes. In some cases, a single physical drive (such as a very large fixed disk) is partitioned into two or more logical drives, each of which is assigned a drive letter.

MS-DOS validates the drive code in a command or filename before carrying out a command. In the default case, MS-DOS recognizes a maximum of five drives (A–E), depending on the total number of default devices and devices incorporated into the system using installable device drivers. (MS-DOS does not consider a drive letter valid unless it refers to a physical or logical device.) The LASTDRIVE command configures MS-DOS to accept additional drive codes, to a total of 26 (A–Z). This also makes it possible to use fictitious drive letters with the SUBST command to assign a drive letter to a subdirectory.

If the letter code for a LASTDRIVE command specifies fewer drives than are physically present in the system (including installed device drivers), MS-DOS uses the actual number of physical drives.

Example

To configure MS-DOS to recognize a maximum of eight logical disk drives, insert the line

```
LASTDRIVE=H
```

into the CONFIG.SYS file and restart the system.

Message

Unrecognized command in CONFIG.SYS
An illegal value was specified in the LASTDRIVE command.

CONFIG.SYS: SHELL

2.0 and later

Specify Command Processor

Purpose

Defines the name and, optionally, the location of the file that contains the operating system's command processor.

Syntax

SHELL=[*drive*:][*path*]*filename* [*options*]

where:

filename is the name of the file containing the command processor, optionally preceded by a drive and/or path.

options specifies any switches and other parameters needed by the designated command processor; the SHELL command itself has no switches.

Description

The command processor, or shell, is the user's interface to the operating system. It is responsible for parsing and carrying out the user's commands, including the loading and execution of other programs from the disk. MS-DOS uses the SHELL command in the CONFIG.SYS file to locate and load the command interpreter for the system during its initialization process.

The default shell for MS-DOS is the file COMMAND.COM. This file is loaded by MS-DOS from the root directory of the system disk if no SHELL command is found in the CONFIG.SYS file or if no CONFIG.SYS file exists.

The most common use of the SHELL command is simply to advise MS-DOS that COMMAND.COM is stored in a location other than the root directory; MS-DOS then sets the COMSPEC variable in the environment block to COMMAND.COM, preceded by the location specified in the SHELL command. (This can be verified by typing the SET command at the command prompt.) Another common use of SHELL is to specify switches or other parameters for COMMAND.COM itself (*see* USER COMMANDS: COMMAND).

Example

To specify the file VISUAL.COM in the root directory of drive C as the system's command processor, insert the line

```
SHELL=C:\VISUAL.COM
```

into the CONFIG.SYS file and restart the system.

Message

Bad or missing command interpreter
The path or filename in the SHELL command is invalid or the file does not exist.

CONFIG.SYS: STACKS

3.2

Configure Internal Stacks

Purpose

Defines the number and size of stacks for system interrupt handlers.

Syntax

STACKS=*number,size*

where:

number is the number of stacks allocated for use by interrupt handlers (8–64, default = 9).

size is the size of each stack in bytes (32–512, default = 128).

Description

Each time certain hardware interrupts occur (02H, 08–0EH, 70H, and 72–77H), MS-DOS version 3.2 switches to an internal stack before transferring control to the handler that will service the interrupt. In the case of nested interrupts, MS-DOS checks to ensure that both interrupts do not get the same stack. After the interrupt has been processed, the stack is released. This protects the stacks owned by application programs or system device drivers from overflowing when several interrupts occur in rapid succession.

The STACKS command configures the number and size of internal stacks available for interrupt handling and thus controls the number of interrupts that can exist only partially processed while still allowing another interrupt to occur.

The *number* parameter sets the number of internal stacks to be allocated; *number* must be in the range 8 through 64. The *size* parameter is the number of bytes allocated per stack frame; *size* must be in the range 32 through 512.

If too many interrupts occur too quickly and the pool of internal stack frames is exhausted, the system halts with the message *Internal Stack Overflow*. Increasing the *number* parameter in the STACKS command usually corrects the problem.

Example

To configure 10 stacks of 256 bytes each for use by interrupt handlers, insert the line

```
STACKS=10,256
```

into the CONFIG.SYS file and restart the system.

Message

Unrecognized command in CONFIG.SYS
An invalid number was specified in the STACKS command.

COPY

1.0 and later

Copy File or Device

Internal

Purpose

Copies one or more files from one disk, directory, or filename to another. Can also copy files to or from character devices.

Syntax

COPY *source* [/A] [/B] [+*source* [/A] [/B]...] [*destination*] [/A] [/B] [/V]

where:

source	is the names of the file(s) to be copied, optionally preceded by a drive and/or path; wildcard characters are permitted in filenames. The source can also be a device.
destination	is the location and, optionally, the name(s) for the copied file(s) and can be preceded by a drive; wildcard characters are permitted in the filename. The destination can also be a device.
/A	indicates that the previous file is an ASCII text file.
/B	indicates that the previous file is a binary file.
/V	performs read-after-write verification of destination file(s).

Description

The COPY command copies one or more source files to one or more destination files. When multiple files are copied, the name of each source file is displayed as it is processed. The COPY command can also be used to send the contents of a file to a character device or to copy input from a character device into a file.

The *source* parameter identifies the file or files to be copied. It can consist of any combination of drive, path, and filename or it can be a device name. If a path without a filename is specified, all files in the named directory are copied. Several source files can be concatenated into a single destination file by placing a + operator between their names; if the source filename contains a wildcard but the destination name does not, all the source files are concatenated into the specified destination.

Warning: When multiple source files are concatenated into a destination file with the same name as one of the source files, that filename should be specified as the *first* source file. Otherwise, the contents of the source file will be destroyed before the file is copied.

When a device is specified as the source, it is usually the console (CON), for copying keyboard input to a file or another device. Keyboard input is terminated by pressing Ctrl-Z or F6 (on IBM PCs or compatibles) and then the Enter key.

The *destination* parameter also can consist of any combination of drive, path, and filename or be a device name. Unless the source files are being renamed as part of the operation, *destination* is usually simply a drive and/or path specifying where to place the copied files. If no destination is specified, the source file is copied to a file with the same name in the current directory of the default disk drive; if the source file in this case is itself in the current directory of the current drive, an error message is displayed and the copy operation is aborted. If files are being concatenated and no destination is specified, the source files are copied sequentially into one file in the current directory with the same name as the first source file. If the first source file already exists, the second file and any additional specified files are appended sequentially to the first source file.

The /A and /B switches control the manner in which the COPY command operates on a file. Both switches affect the file specification immediately preceding them and any subsequent file specifications in the command until another /A or /B switch is encountered, at which point the new /A or /B switch takes effect for the file immediately preceding it and for any subsequent files.

The /A switch indicates that a file is an ASCII text file. When the /A switch is applied to a source file, the file is copied up to, but not including, the first Control-Z (^Z) character in the file. When the /A switch is applied to a destination file, a Control-Z character is appended by the COPY command as the last character of the new file.

The /B switch indicates a binary file. When /B is applied to a source file, the exact number of bytes in the original file are copied without regard to Control-Z or any other control characters. When the /B switch is applied to a destination file, no Control-Z character is appended to the newly created file.

The default values for the /A and /B switches for file-to-file copies are /A when source files are being concatenated and /B otherwise. When a file is being copied to or from a character device, the /A switch is the default.

The /V switch causes a read-after-write verification of each block of the destination file. Its effect is equivalent to that of the VERIFY ON command. No comparison is made between the source and destination files—the /V switch simply causes MS-DOS to verify that the destination file has been written correctly.

Examples

To copy the file REPORT.TXT from the root directory of the disk in drive B to a file named FINAL.RPT in the \WP\DOCS directory on the current drive, type

```
C>COPY B:\REPORT.TXT \WP\DOCS\FINAL.RPT  <Enter>
```

To make a copy of the file A:\V2\SOURCE\MENUMGR.C in the current directory of the current drive, type

```
C>COPY A:\V2\SOURCE\MENUMGR.C  <Enter>
```

To copy all files with the extension .DOC in the current directory of the disk in drive A to files with the same filenames but a .TXT extension in the current directory of the current drive, type

```
C>COPY A:*.DOC *.TXT  <Enter>
```

To combine the files PROLOG.C, MENUMGR.C, and EPILOG.C in the current directory of the current drive into a single file named VISUAL.C in the current directory of the current drive, type

```
C>COPY PROLOG.C+MENUMGR.C+EPILOG.C VISUAL.C  <Enter>
```

To append the files MENUMGR.C and EPILOG.C to an existing file named PROLOG.C in the current directory of the current drive, type

```
C>COPY PROLOG.C+MENUMGR.C+EPILOG.C  <Enter>
```

To copy the file MENUMGR.MAP in the current directory of the current drive to the system printer, type

```
C>COPY MENUMGR.MAP PRN  <Enter>
```

To copy input from the keyboard (CON) to a file named MENU.BAT in the current directory of the current drive, type

```
C>COPY CON MENU.BAT  <Enter>
```

Text subsequently entered from the keyboard is placed into the file MENU.BAT until a Ctrl-Z or F6 is pressed.

To copy all files in the \MEMOS directory on the current drive to the \ARCHIVE directory on the disk in drive B, type

```
C>COPY \MEMOS\*.* B:\ARCHIVE  <Enter>
```

or

```
C>COPY \MEMOS B:\ARCHIVE  <Enter>
```

Messages

***n* File(s) copied**
This informational message is displayed at the completion of a COPY command and indicates the total number of source files processed.

Cannot do binary reads from a device
The COPY command specified a copy from a character device in binary mode. Reenter the command without a /B switch.

Content of destination lost before copy
One of the source files specified as a destination file was overwritten prior to completion of the copy. When the destination name is the same as one of the source names, that file should be specified as the first source file.

File cannot be copied onto itself
The source directory and filename of a file being copied are the same as the destination directory and filename.

File not found
A file specified in the COPY command is invalid or does not exist.

Invalid directory
A directory specified in the COPY command is invalid or does not exist.

CTTY

2.0 and later

Assign Standard Input/Output Device

Internal

Purpose

Specifies the character device to be used as standard input and output.

Syntax

CTTY *device*

where:

device is the logical character-device name.

Description

MS-DOS ordinarily uses the computer's built-in keyboard and screen (CON) as standard input and output. The CTTY command allows another character device to be assigned instead.

CTTY allows MS-DOS commands to be issued from a terminal attached to the computer's serial port or from another custom device with a screen and keyboard. Although PRN and NUL are valid MS-DOS device names, they should not be used with this command, as they have no input capability.

Programs that do not use MS-DOS function calls to perform their input and output will not be affected by the CTTY command. Microsoft BASIC is an example of such a program.

Examples

To use a terminal connected to the serial port as standard input and output for programs, type

```
C>CTTY AUX  <Enter>
```

To reinstate the normal keyboard and video display (CON) as standard input and output for programs, type

```
C>CTTY CON  <Enter>
```

on the currently assigned console device.

Message

Invalid device
The specified device is not a legal character-device name or does not exist in the system.

DATE

1.0 and later

Set Date

Internal

Purpose

Sets or displays the system date.

Syntax

DATE *mm-dd-yy*

or

DATE *mm/dd/yy*

or

DATE *mm.dd.yy* (versions 3.0 and later)

where:

mm	is the month (1–12).
dd	is the day (1–31).
yy	is the year (80–99 or 1980–1999; 80–79 or 1980–2079 with versions 3.0 and later).

Description

All computers that run MS-DOS have as part of their hardware configuration a timer, or clock, that maintains the current system date and time. Among other uses, the current date and time are inserted into a file's directory entry when the file is created or modified.

The DATE command allows the user to display or modify the current date that is being maintained by the system's real-time clock. The command is executed automatically by MS-DOS when the system is initialized, unless there is an AUTOEXEC.BAT file on the system disk, in which case DATE is executed only if it is included in the file.

A date entered using the DATE command does not permanently change the system date; the newly entered date will be lost when the system is turned off or reset. On IBM PC/ATs and compatibles, which have a built-in battery-backed clock/calendar, the system setup program (found on the Diagnostics for IBM Personal Computer AT disk or equivalent) must be used to permanently alter the date stored in the machine. On IBM PCs, PC/XTs, and compatibles equipped with add-on cards containing battery-backed clock/calendar circuitry, it is generally necessary to run a time/date installation program (included with the card) when the system is turned on to set the system date and time from the clock/calendar on the card. The DATE command usually has no effect on these card-mounted clock/calendars.

The order of the day, month, and year in the DATE command depends on the country code, which is set with the COUNTRY command in the CONFIG.SYS file. The format shown here is for the USA.

Examples

To set the system date to October 15, 1987, type

```
C>DATE 10-15-87  <Enter>
```

or

```
C>DATE 10/15/87  <Enter>
```

or

```
C>DATE 10.15.87  <Enter>
```

To display the current system date, type

```
C>DATE  <Enter>
```

and MS-DOS will respond in the form

```
Current date is Thu 10-15-1987
Enter new date (mm-dd-yy):
```

To leave the date unchanged, press the Enter key.

Messages

Current date is *day mm-dd-yyyy*
Enter new date (mm-dd-yy):
This informational message and prompt are displayed when MS-DOS is started and there is no AUTOEXEC.BAT file on the system disk, when the DATE command is entered alone, or when the DATE command is included in the AUTOEXEC.BAT file.

Invalid date
Enter new date (mm-dd-yy):
The date entered in the command line or in response to the prompt from the DATE command was not formatted properly or was invalid.

DEL or ERASE

1.0 and later

Delete File

Internal

Purpose

Deletes a file or set of files. DEL and ERASE are synonymous.

Syntax

DEL [*drive*:][*path*]*filename*

or

ERASE [*drive*:][*path*]*filename*

where:

filename is the name of the file(s) to be deleted, optionally preceded by a drive and/or path; wildcard characters are permitted in the filename.

Description

The DEL command marks the directory entry for the specified file as deleted and frees the disk sectors occupied by the file. If the command line ends with *.* or a directory name (including the special directory names . and ..), MS-DOS prompts the user for confirmation before deleting all the files in the current or specified directory. Note that in the case of a directory name, the directory itself is not removed; only the files within it are deleted.

Warning: If the filename specification begins with an * wildcard and the extension is also * (for example, *xyz.*), DEL interprets the specification as *.* and prompts the user for confirmation before deleting all files from the current or specified directory.

Examples

To delete the file HELLO.C from the current directory on the current drive, type

```
C>DEL HELLO.C  <Enter>
```

To delete all files with the extension .OBJ from the \SOURCE directory on the disk in drive D, type

```
C>DEL D:\SOURCE\*.OBJ  <Enter>
```

To delete all files from the current directory on the current drive, type

```
C>DEL *.*  <Enter>
```

or

```
C>DEL .  <Enter>
```

In this case, MS-DOS will prompt for confirmation that all files should be deleted.

To delete all files from the directory \WORD\LETTERS on the current drive, type

```
C>DEL \WORD\LETTERS  <Enter>
```

Again, MS-DOS will prompt for confirmation that all files should be deleted.

Messages

Access denied
The specified file is read-only. Use the ATTRIB command with the –R switch to remove the file's read-only status.

Are you sure (Y/N)?
This message prompts the user for confirmation if the command would delete all files in a directory (if the command line ends with a directory name or *.*). Respond with *Y* to delete all files in the directory; respond with *N* to terminate the command.

File not found
The filename in the command is invalid or the file does not exist in the specified directory.

Invalid directory
One of the directories named in the file specification is invalid or does not exist.

Invalid drive specification
The drive code in the file specification is invalid or the named drive does not exist in the system.

DIR

1.0 and later

Display Directory

Internal

Purpose

Displays a list of a directory's files and subdirectories.

Syntax

DIR [*drive*:][*path*][*filename*] [/P] [/W]

where:

filename	is the name of the file, optionally preceded by a drive and/or path, whose directory entry is to be displayed; wildcard characters are permitted.
/P	causes a pause after each screen page of display.
/W	causes a wide display of filenames formatted five across.

Description

The DIR command displays information about the files in a directory. It also displays information about the volume name of the disk that contains the directory, the total number of files and subdirectories in the directory, and the amount of free space remaining on the disk.

The normal format of the DIR command's output is

```
 Volume in drive C is HARDDISK
 Directory of  C:\ASM

.            <DIR>       9-19-85   7:09p
..           <DIR>       9-19-85   7:09p
LIB          <DIR>       9-17-86  11:31p
SOURCE       <DIR>       9-17-86  11:31p
AT86     EXE    41146    5-13-85   5:18p
CREF     EXE    15028   10-16-85   4:00a
DEBUG    COM    15552    3-07-85   1:43p
EXE2BIN  EXE     2816    3-07-85   1:43p
EXEMOD   EXE    11034   10-16-85   4:00a
EXEPACK  EXE    10848   10-16-85   4:00a
LIB      EXE    28716   10-16-85   4:00a
LINK     EXE    43988   10-16-85   4:00a
MAKE     EXE    24300   10-16-85   4:00a
MAPSYM   EXE    18026   10-16-85   4:00a
MASM     EXE    85566   10-16-85   4:00a
SYMDEB   EXE    37021   10-16-85   4:00a
T86      EXE    49024   12-06-84   4:03p
       17 File(s)    4022272 bytes free
```

The first line shows the volume label of the disk that contains the directory being displayed; the second line gives the full pathname of the directory. The subsequent lines are

the names of the files and subdirectories within the current or specified directory. Each entry includes the time and date the file or subdirectory was created or last modified.

Files are shown with their exact size in bytes; directories are shown with the symbol <DIR>. If the directory being listed is not the root directory of the disk, it always contains the two special directory entries . and .., which are aliases for the current directory and the parent directory, respectively. These aliases are included in the total file count in the last line of the display.

Subsets of the files and subdirectories in the current or specified directory of the current or specified drive can be listed by including a filename with wildcards in the command line. For example, the filename *.DOC will cause DIR to list only the files with a .DOC extension.

If the command line ends with a drive or path, DIR automatically appends an *.*, causing all files and subdirectories in the current or specified directory of the current or specified drive to be listed. If a filename is included but no extension is given, DIR appends a .* to the filename, causing all files with that name to be listed, regardless of their extension. If a filename ending with a . is included, nothing is appended and all matching subdirectories and filenames without extensions are listed.

The /P switch causes a pause in the display after each screen page (23 lines plus a message). The listing resumes when the user presses a key.

The /W switch causes the list to be in a more compact format by omitting size and date/time information and by displaying the filenames five across:

```
 Volume in drive C is HARDDISK
 Directory of  C:\ASM
.             ..            LIB           SOURCE        AT86     EXE
CREF     EXE  DEBUG    COM  EXE2BIN  EXE  EXEMOD   EXE  EXEPACK  EXE
LIB      EXE  LINK     EXE  MAKE     EXE  MAPSYM   EXE  MASM     EXE
SYMDEB   EXE  T86      EXE
       17 File(s)   4022272 bytes free
```

When the /W form of the listing is displayed, subdirectories are not easily distinguished from files because the <DIR> symbol is not shown.

Examples

To list all files in the current directory on the current drive, type

```
C>DIR  <Enter>
```

To list all files in the current directory on the disk in drive B, type

```
C>DIR B:  <Enter>
```

or

```
C>DIR B:*.*  <Enter>
```

To list all files in the directory \SOURCE on the current drive, type

```
C>DIR \SOURCE  <Enter>
```

or

```
C>DIR \SOURCE\*.*  <Enter>
```

To list all files with the extension .OBJ in the \LIB directory on the disk in drive D, type

```
C>DIR D:\LIB\*.OBJ  <Enter>
```

To list all files in the parent directory of the current directory on the current drive, type

```
C>DIR ..  <Enter>
```

To list all files in the current directory on the current drive, sorted by filename and extension, type

```
C>DIR ¦ SORT  <Enter>
```

To list all files in the current directory on the current drive, sorted by extension, type

```
c>DIR ¦ SORT /+10  <Enter>
```

The */+10* instructs SORT to sort the directory entries starting at the tenth column, which is the first column of the filename extension.

To list the subdirectories and files without extensions in the current directory, type

```
C>DIR *.  <Enter>
```

To print the directory on an attached printer instead of displaying it on the screen, type

```
C>DIR > PRN  <Enter>
```

To make a copy of the directory in a file called FILES.TXT, type

```
C>DIR > FILES.TXT  <Enter>
```

Messages

File not found
A filename was included in the command line and no matching files were found.

Invalid directory
An element of the path included in the command line does not exist.

Invalid drive specification
The specified drive is invalid or is not present in the system.

Strike a key when ready...
If the DIR command includes the /P switch, the display is suspended after each 23 lines and this message prompts the user to press a key to see the next screenful of entries.

DISKCOMP

3.2

Compare Floppy Disks

External

Purpose

Compares two entire floppy disks on a sector-by-sector basis and reports any differences. This command was included with PC-DOS beginning with version 1.0. To compare individual files, *see* USER COMMANDS: COMP; FC.

Syntax

DISKCOMP [*drive1*:] [*drive2*:] [/1] [/8]

where:

drive1	is the drive containing the first disk to be compared.
drive2	is the drive containing the second disk to be compared.
/1	compares only the first sides of the disks.
/8	compares only the first eight sectors of each track.

Description

The DISKCOMP command compares the physical sectors of one floppy disk with those of another. The *drive1* and *drive2* parameters designate the drives holding the two disks to be compared; the drives should always be of the same type. If *drive2* is omitted, DISKCOMP uses the current drive. If both *drive1* and *drive2* are omitted or are identical, DISKCOMP performs the comparison using a single drive, prompting the user to swap disks as required.

Ordinarily, DISKCOMP determines the disk format by inspecting the disk in *drive1*. The /1 and /8 switches override this check so that only one side of the disks or only the first eight sectors of each track are compared, regardless of the actual format of the disks.

If all the sectors on all the tracks are identical, DISKCOMP displays the message *Compare OK*. If differences are found, DISKCOMP reports them by issuing a message that includes the numbers of the track and disk side (read/write head) where the differences occur. Because DISKCOMP works at the level of the disks' physical sectors and is ignorant of the control areas and file structures imposed on a disk by MS-DOS, it also reports as errors bad sectors that were marked during the FORMAT process.

When DISKCOMP finishes comparing two disks, it displays a prompt that allows the user to choose between comparing another pair of disks and returning to the MS-DOS command level.

DISKCOMP cannot be used with a network drive or with a drive created or affected by an ASSIGN, JOIN, or SUBST command, nor can it be used with fixed disks.

Return Codes

0 Compared disks were identical.
1 Differences were found between the compared disks.
2 DISKCOMP was terminated with a Control-C.
3 Bad sector was found on one of the disks being compared.
4 Initialization error was encountered: not enough memory, syntax error in command line, or invalid drive specified in command line.

Note: Return codes are not present in the PC-DOS version of DISKCOMP.

Examples

To compare the disk in drive A with the disk in drive B, type

```
C>DISKCOMP A: B:  <Enter>
```

To compare two disks using only drive A, type

```
C>DISKCOMP A: A:  <Enter>
```

To compare only the first side of the disk in drive A with the first side of the disk in drive B, type

```
C>DISKCOMP A: B: /1  <Enter>
```

To compare only the first eight sectors of each track on one side of one disk with the first eight sectors of each track on one side of another disk using only drive A, type

```
C>DISKCOMP A: A: /1 /8  <Enter>
```

Messages

Cannot DISKCOMP to or from an ASSIGNed or SUBSTed drive
One of the specified drives has been affected by an ASSIGN or SUBST command.

Cannot DISKCOMP to or from a network drive
One of the specified drives is a network device.

Compare another diskette (Y/N) ?
This prompt allows comparison of another pair of disks. Respond with *Y* to cause DISKCOMP to prompt for insertion of the next pair of disks to be compared; respond with *N* to exit to MS-DOS.

Compare error on side *n*, track *n*
A difference was detected between the two disks being compared.

Compare OK
The two disks being compared are identical.

Compare process ended
The disk comparison was terminated as the result of a fatal error.

Comparing *n* tracks,
***n* sectors per track, *n* side(s)**
This informational message specifies the format of the two disks being compared.

DEVICE Support Not Present
The disk drive does not support MS-DOS 3.2 device control.

Drive *X* not ready
Make sure a diskette is inserted into
the drive and the door is closed
DISKCOMP was unable to read the disk in the specified drive.

Drive types or diskette types
not compatible
Single-sided disks cannot be compared with double-sided disks, nor high-density disks with double-density disks.

FIRST diskette bad or incompatible
DISKCOMP is unable to determine the format of the first disk.

Incorrect DOS version
The version of DISKCOMP is not compatible with the version of MS-DOS that is running.

Insert diskette with directory that contains
COMMAND.COM in drive *X* and strike any key when ready
If the system was booted from a floppy disk and the system disk was then removed in order to use DISKCOMP, the user must replace the system disk after the compare operation is complete.

Insert FIRST diskette in drive *X*:
Press any key when ready...
This message prompts the user to insert the first disk of a pair to be compared.

Insert SECOND diskette in drive *X*:
Press any key when ready...
This message prompts the user to insert the second disk of a pair to be compared.

Insufficient memory
The available system memory is insufficient to load and execute the DISKCOMP program.

Invalid drive specification
Specified drive does not exist
or is non-removable
One of the drives specified in the command line is invalid or does not exist.

Invalid parameter
Do not specify filename(s)
Command format: DISKCOMP d: d: [/1][/8]
A syntax error was detected in the command line, usually caused by an incorrect switch.

SECOND diskette bad or incompatible
The second disk of a pair to be compared does not have the same format as the first disk or has bad sectors preventing DISKCOMP from determining its format.

Unrecoverable read error on drive *X*:
The disk in the specified drive contains an unreadable sector.

DISKCOPY

2.0 and later

Copy Floppy Disks

External

Purpose

Performs a sector-by-sector copy of one entire floppy disk to another floppy disk. This command was included with PC-DOS beginning with version 1.0. To copy individual files, *see* USER COMMANDS: COPY.

Syntax

DISKCOPY [*drive1*:] [*drive2*:] [/1]

where:

drive1	is the drive containing the disk to be copied.
drive2	is the drive containing the disk that will become the copy.
/1	copies only the first side of the disk in *drive1* (MS-DOS version 3.2).

Description

The DISKCOPY command duplicates a floppy disk, performing the copy on a physical sector-by-sector basis. The *drive1* parameter specifies the location of the disk to be copied (the source disk). The *drive2* parameter specifies the location of the disk that will become the copy (the destination disk). If *drive2* is omitted, the current drive is used as the destination drive; if both *drive1* and *drive2* parameters are omitted or are the same, DISKCOPY performs the copy operation using a single drive, prompting the user to swap the disks as necessary.

DISKCOPY examines the destination disk before writing any information and terminates with an error message if it does not have the same format as the source disk. If the destination disk is not formatted, DISKCOPY formats it with the same format as the source disk, as part of the DISKCOPY operation.

Note: With MS-DOS versions 2.0 through 3.1, the destination disk must be formatted using the FORMAT command before DISKCOPY can be used. All PC-DOS versions of DISKCOPY will automatically format the destination disk, if necessary.

When DISKCOPY finishes copying a disk, it displays a prompt that allows the user to choose between copying another disk and returning to the MS-DOS command level.

Because DISKCOPY creates an exact duplicate of the source disk, any file fragmentation present on the source disk is also present on the destination disk after the DISKCOPY process is complete. To eliminate fragmentation of the source files, they should be copied to the destination disk individually using COPY or XCOPY.

The DISKCOPY command cannot be used with a network drive or with a drive created or affected by an ASSIGN, JOIN, or SUBST command, nor can it be used with fixed disks.

Return Codes

0 Disk was copied successfully.
1 Nonfatal but unrecoverable read or write error occurred (no Interrupt 24H generated).
2 DISKCOPY was terminated with a Control-C.
3 Fatal error was encountered: unreadable source disk or unformattable destination disk.
4 Initialization error was encountered: not enough memory, syntax error in command line, or invalid drive specified in command line.

Note: Return codes are not present in the PC-DOS version of DISKCOPY.

Examples

To copy the contents of the disk in drive A to the disk in drive B, type

```
C>DISKCOPY A: B:  <Enter>
```

To copy the contents of the disk in drive A using only one drive, type

```
C>DISKCOPY A: A:  <Enter>
```

To copy only the first side of the disk in drive A to the first side of the disk in drive B, type

```
C>DISKCOPY A: B: /1  <Enter>
```

Messages

Cannot DISKCOPY to or from
an ASSIGNed or SUBSTed drive
One of the specified drives has been affected by an ASSIGN or SUBST command.

Cannot DISKCOPY to or from
a network drive
One of the specified drives is a network device.

Copy another diskette (Y/N) ?
This prompt allows copying of another disk. Respond with *Y* to cause DISKCOPY to prompt for insertion of the next set of disks; respond with *N* to exit to MS-DOS.

Copying *n* tracks
***n* sectors per track, *n* side(s)**
This informational message specifies the format of the source disk being copied.

Copy process ended
The DISKCOPY process has been successfully completed or has been terminated by a fatal error. In the latter case, this message is preceded by another message explaining the error.

DEVICE Support Not Present
The disk drive does not support MS-DOS version 3.2 device control.

Disk error while reading drive *X*:
Abort, Retry, Ignore?
A bad sector was detected on the source disk. This does not necessarily invalidate the disk copy; the bad sector may originally have been detected and flagged by the FORMAT program and therefore not included in any file. One solution is to copy the files individually using the COPY command.

Drive *X*: not ready
Make sure a diskette is inserted into
the drive and the door is closed
DISKCOPY was unable to read the disk in the specified drive.

Drive types or diskette types
not compatible
Single-sided disks cannot be copied to or from double-sided disks, nor high-density disks to or from double-density disks.

Formatting while copying
The destination disk was not previously formatted. It is given the same format as the source disk as part of the DISKCOPY operation (MS-DOS version 3.2).

Incorrect DOS version
The version of DISKCOPY is not compatible with the version of MS-DOS that is running.

Insert diskette with directory that contains
COMMAND.COM in drive *X* and strike any key when ready
If the system was booted from a floppy disk and the system disk was then removed in order to use DISKCOPY, the user must replace the system disk after the copy operation is complete.

Insert SOURCE diskette in drive *X*:
Press any key when ready...

or

Insert TARGET diskette in drive *X*:
Press any key when ready...
These messages prompt the user to insert the source and destination disks before beginning the copy operation.

Insufficient memory
The available system memory is insufficient to load and execute the DISKCOPY program.

Invalid drive specification
Specified drive does not exist,
or is non-removable
One of the drives specified in the command line is invalid or does not exist. A fixed disk cannot be the source or destination disk for a DISKCOPY operation.

Invalid parameter
Do not specify filename(s)
Command Format: DISKCOPY d: d: [/1]
A syntax error was detected in the command line, usually caused by an incorrect switch or by the use of a filename instead of (or in addition to) a disk drive.

SOURCE diskette bad or incompatible

or

TARGET diskette bad or incompatible
The source disk could not be read or the destination disk could not be formatted.

Target diskette is write protected
The destination disk has a write-protect tab on it.

Target diskette may be unusable
Unrecoverable read or write errors were encountered while copying the source disk to the destination disk. The newly copied disk may not be an accurate copy.

Unrecoverable read error on drive *X*:
side *n*, track *n*

or

Unrecoverable write error on drive *X*:
side *n*, track *n*
The disk in the specified drive contained a sector that could not be successfully read or written.

DRIVER.SYS

3.2

Configurable External-Disk-Drive Driver

External

Purpose

Installs and configures external disk drives or assigns logical drive letters to existing floppy-disk drives.

Syntax

DEVICE=DRIVER.SYS /D:*n* [/C] [/F:*n*] [/H:*n*] [/N] [/S:*n*] [/T:*n*]

where:

/D:*n* is the drive number (0–127 for floppy disks, 128–255 for fixed disks) and must always be the first switch in the command line.

/C specifies that door-lock-status support is available.

/F:*n* is the form-factor index for the device (default = 2):

0	320/360 KB
1	1.2 MB
2	720 KB
3	8" single-density floppy disk
4	8" double-density floppy disk
5	fixed disk
6	magnetic-tape drive
7	other

/H:*n* is the number of heads supported by the disk drive (1–99).

/N specifies a nonremovable block device.

/S:*n* is the number of sectors per track (1–40).

/T:*n* is the tracks per read/write head (1–999).

Description

When the computer is turned on or restarted, MS-DOS assigns numbers to all existing internal disk drives. The DRIVER.SYS file — an installable, configurable block-device driver for external disk drives and other mass-storage devices — allows installation of peripheral devices that are not supported by the resident drivers in the MS-DOS BIOS module. DRIVER.SYS can also assign a logical drive letter to an existing disk drive, thus giving the device two drive letters. (This allows such activities as copying files between like media — for example, copying files from one 1.2 MB 5.25-inch disk to another — using the same drive.)

The /D:*n* switch assigns a unit number to the additional disk drive or specifies the number of the existing disk drive that is to be assigned a logical drive letter. (Floppy-disk unit numbers begin at 0; fixed-disk numbers begin at 80H.) For example, if the system contains two floppy-disk drives (0 and 1), an external floppy-disk drive requiring DRIVER.SYS would be assigned the value 2; MS-DOS would then assign that drive the next available drive letter. If the number used with the /D:*n* switch references an existing drive (for example, 0, the first floppy-disk drive), MS-DOS assigns the drive the next available drive letter, allowing the one drive unit to be referenced by two drive letters. The /D:*n* switch is not optional and must precede all other switches in the command line.

The /C, /F:*n*, and /N switches describe characteristics of the disk drive that is being selected for use with DRIVER.SYS. The /C switch is included only if the device has a status line indicating whether the disk in the drive has been changed. (This information is used by the driver to optimize disk accesses to the directory and file allocation table.) If the device does not have a status line, /C will have no effect. The /F:*n* option describes the form-factor index used by the device. The permissible values for *n* are given in the preceding table; the default type is a 720 KB disk. The /N switch indicates that the block device is nonremovable. Access to such devices is more efficient than access to removable media because MS-DOS can eliminate calls to the driver for a media-change check.

The /H:*n*, /S:*n*, and /T:*n* switches describe the physical layout of the recording medium. /H:*n* specifies the number of recording surfaces, or read-write heads, supported by the drive (1–99). /S:*n* is the number of sectors per track (1–40) and /T:*n* is the tracks per side (1–999). (The total number of physical sectors on a given disk is found by multiplying the number of heads by the tracks per side and the sectors per track.)

Note: The values used with these switches must be supported by the device being installed. If DRIVER.SYS is used to assign a logical drive letter to an existing physical device, the values used with the switches must be identical to the characteristics imposed by the default device driver.

Examples

To install a driver for an external 720 KB disk drive in a system that already has two 5.25-inch floppy-disk drives, insert the line

```
DEVICE=DRIVER.SYS /D:02
```

into the CONFIG.SYS file and restart the system.

Assume that an IBM PC/AT or compatible has three disk drives installed: Drive A is a 1.2 MB 5.25-inch floppy-disk drive; drive B is a 360 KB 5.25-inch floppy-disk drive; drive C is a 30 MB fixed-disk drive. To assign the logical drive letter D to the existing drive A, effectively giving the one drive two drive letters, insert the line

```
DEVICE=DRIVER.SYS /D:0 /F:1 /H:2 /S:15 /T:80 /C
```

into the CONFIG.SYS file and restart the system.

Messages

Bad or missing DRIVER.SYS
The file DRIVER.SYS could not be found in the root or specified directory or has been damaged.

ERROR - Incorrect DOS version
The version of DRIVER.SYS is not compatible with the version of MS-DOS that is running.

ERROR - No drive specified
The /D:*n* switch was not included in the command line.

Loaded External Disk Driver for Drive *X*
The device driver has been successfully installed and this message informs the user of the drive letter assigned to the device.

Sector size too large in file DRIVER.SYS
DRIVER.SYS uses a sector size that is larger than the sector size used by any of the system's default disk drivers. The driver cannot be used because MS-DOS's internal disk buffers will not be large enough to hold a sector read from the device.

EDLIN

Line Editor

1.0 and later

External

Purpose

Creates and changes ASCII text files.

Syntax

EDLIN [*drive*:][*path*]*filename* [/B]

where:

filename	is the name of an ASCII text file to be created or edited, optionally preceded by a drive and/or path.
/B	causes logical end-of-file marks within the file to be ignored (versions 2.0 and later).

Description

The EDLIN program is a simple line-oriented editor that can be used to create or maintain short text files. The user references and edits text by line number; EDLIN displays these numbers for convenience but they do not become part of the file. Each line of the file being edited can be a maximum of 253 characters.

The *filename* parameter specifies a plain ASCII text file; if the file does not already exist, EDLIN creates it. (EDLIN cannot be used on most files created by word-processing programs because such document files have embedded formatting codes and other formatting information that EDLIN cannot interpret.) EDLIN does not assume any extensions; the user must type the complete filename. (EDLIN does not permit editing of a .BAK file.)

If *filename* is a previously existing text file, EDLIN loads lines from the file into memory until the editing buffer is 75 percent full or until a logical end-of-file mark or the physical end of the file is reached. The /B switch forces EDLIN to ignore any logical end-of-file marks (1AH, or Control-Z) the file may contain. If the file is too large for the edit buffer, the Write Lines to Disk (W) and Append Lines from Disk (A) commands are used during the edit session to process the remaining portions of the file.

Once the file is created or loaded into the editing buffer, EDLIN displays its asterisk prompt (*) and the user can begin entering editing commands.

EDLIN commands consist of a single character, in either uppercase or lowercase, usually preceded by one or more line numbers. More than one command can be entered on a single line by separating the commands with semicolons. EDLIN does not execute a command until the Enter key is pressed.

The EDLIN commands are

Command	Action
linenumber	Edit line.
A	Append lines from disk.
C	Copy lines (versions 2.0 and later).
D	Delete lines.
E	End editing session.
I	Insert lines.
L	List lines.
M	Move lines (versions 2.0 and later).
P	Display in pages (versions 2.0 and later).
Q	Quit without saving changes.
R	Replace text.
S	Search for text.
T	Transfer another file into the edit buffer (versions 2.0 and later).
W	Write lines to disk.

Each of these commands is discussed in detail in the following pages.

All EDLIN commands that accept a line number or range of line numbers can also recognize the following symbolic references:

Symbol	Meaning
#	The line after the last line in the edit buffer
.	The current line
+*n* or −*n*	A line number relative to the current line (for example, +5 = five lines past the current line)

When the user terminates the editing session with the E command, EDLIN gives the new file the same name as the original file and renames the original (unchanged) file with the extension .BAK. Any previous file with the same name and the extension .BAK is lost. When the user terminates the editing session with the Q command, the original filename remains unchanged.

Example

To edit the file AUTOEXEC.BAT in the root directory of the current drive, type

```
C>EDLIN \AUTOEXEC.BAT  <Enter>
```

Messages

Cannot edit .BAK file — rename file
Files with the extension .BAK cannot be edited with EDLIN. Rename the file or copy it to a file with the same name but a different extension.

End of input file
The entire file has been read into memory.

File is READ-ONLY
Files marked with the read-only attribute cannot be edited. Remove the read-only attribute with the ATTRIB command or copy the file to a file with a different name.

File name must be specified
The command line did not include a filename.

File not found
The file named in the command line could not be found or does not exist.

Incorrect DOS version
The version of EDLIN is not compatible with the version of MS-DOS that is running.

Insufficient memory
Not enough memory is available to carry out the requested command.

Invalid drive or file name
The command line included a drive that is invalid or does not exist in the system or the filename is not valid.

Invalid Parameter
The command line contained an illegal switch or other invalid parameter.

New file
The file named in the command line did not previously exist. The file is created and the edit buffer is emptied.

Read error in: *filename*
MS-DOS was unable to read the entire file. Run CHKDSK to determine whether the file or disk has been damaged.

EDLIN: *linenumber*

1.0 and later

Edit Line

Purpose

Selects a line of text for editing.

Syntax

linenumber

where:

linenumber is the number assigned by EDLIN to the text line to be edited (1–65534).

Description

The command to edit a particular line of text is simply the line's number or one of the special symbols or expressions that evaluate to a line number, followed by the Enter key. EDLIN displays the current contents of the specified line and copies them to a special editing buffer called the template, then moves the cursor to a new line and displays a prompt in the form of the line number followed by a colon and an asterisk. If a line number is not specified (that is, if the Enter key alone is pressed in response to the EDLIN prompt), EDLIN displays the line following the current line and makes it the current line.

The user can change the text of the specified line by simply entering new text followed by a press of the Enter key, leave the text unchanged by pressing Enter alone, or modify the text by using special editing keys to change a portion of the text that has been placed in the template. These editing keys and their actions are

Key	Action
F1	Copies one character from the template to the new line.
F2*char*	Copies all characters up to the specified character from the template to the new line.
F3	Copies all remaining characters in the template to the new line.
Del	Does not copy (skips over) one character.
F4*char*	Does not copy (skips over) all characters up to the specified character.
Esc	Restarts editing for the current line, leaving the template unchanged.
Ins	Enters/exits character-insert mode.
F5	Makes the newly edited line the new template.
→	Copies one character from the template to the new line.
←	Deletes one character from the new line.
Backspace	Deletes one character from the new line.

Note: Computers that are not IBM-compatible may use a different set of editing keys to perform these actions.

Control characters (those characters with ASCII codes in the range 0–1FH) cannot be inserted into text with the usual Control-key combinations. Instead, the user must press the sequence Ctrl-V, followed by an uppercase character or symbol. For example, Ctrl-C (ASCII code 03H) is entered into text by pressing Ctrl-V followed by a capital C; the Escape character (ASCII code 1BH) is generated by pressing Ctrl-V followed by a left square-bracket character ([).

Examples

To edit line 4, type

```
*4  <Enter>
```

To edit the line two lines ahead of the current line, type

```
*+2  <Enter>
```

EDLIN: A

1.0 and later

Append Lines from Disk

Purpose

Reads lines from the file being edited into the edit buffer.

Syntax

[*n*]A

where:

n is the number of lines to be read from the file.

Description

If the file being edited is too large to fit into the edit buffer, EDLIN ordinarily reads only enough text to fill 75 percent of the buffer when it opens the file, reserving 25 percent of the buffer for additions and changes to the text. The user must then employ the Write Lines to Disk (W) and Append Lines from Disk (A) commands to write and read successive blocks of text until the entire file has passed through the edit buffer.

The A command alone has no effect if the edit buffer is 75 percent or more full. The W command must be used to write lines to the output file and delete them from the buffer; then the A command can read new lines from the input file and append them to the end of the text remaining in the buffer.

The *n* parameter specifies the number of lines to be read from the file. If *n* is omitted or is too large, EDLIN reads only enough lines to fill the editing buffer to 75 percent of its capacity.

Examples

To append 200 lines from the disk file to the edit buffer, type

```
*200A  <Enter>
```

To append as many lines from the file as possible (until the edit buffer is 75 percent full), type

```
*A  <Enter>
```

Message

End of input file
The last section of the file being edited has been read into the edit buffer.

EDLIN: C

2.0 and later

Copy Lines

Purpose

Copies one or more lines from one location in the edit buffer to another.

Syntax

[*first*],[*last*],*destination*[,*count*]C

where:

first	is the number of the first line to be copied.
last	is the number of the last line to be copied.
destination	is the number of the line before which the copied lines are to appear.
count	is the number of times to execute the copy operation.

Description

The Copy Lines (C) command copies one or more text lines, inserting the copied lines at another location in the edit buffer. The original lines that were copied are unchanged. EDLIN then renumbers the edit buffer and makes the first copied line at the destination the new current line.

The *first* and *last* line-number parameters define the block of lines to be copied. (Note that the first line number must be less than or equal to the last line number.) Either or both of these numbers can be omitted (in which case the current line number is used), but the commas must still be entered as placeholders. The *destination* parameter specifies the line before which the copied lines are to be inserted; it is not optional and must not fall within the range of line numbers specified by *first* and *last*. One of the special symbols . (current line) or # (end of buffer) or an expression relative to the current line number (+*n* or −*n*) can be used instead of absolute line numbers.

To replicate the line or lines multiple times, the copy operation can be repeated automatically with the optional parameter *count*. The default value for *count* is one.

Examples

If the current line is line 10, to copy lines 10 through 15 and place the copied lines before line 5, type

```
*10,15,5C  <Enter>
```

or

```
*,15,5C  <Enter>
```

or

```
*,+5,-5C  <Enter>
```

If the current line is line 10, to place three copies of lines 10 through 15 before line 1, type

```
*10,15,1,3C  <Enter>
```

or

```
*,15,1,3C  <Enter>
```

or

```
*,+5,1,3C  <Enter>
```

Messages

Entry error
The command line contained an error such as a first line number that was greater than the last line number or a destination line number that fell within the range *first,last.*

Insufficient memory
The edit buffer does not have sufficient room for EDLIN to carry out the specified command.

Must specify destination line number
No destination line number was specified in the command line; therefore, no changes were made to the edit buffer.

EDLIN: D

1.0 and later

Delete Lines

Purpose

Deletes one or more lines from the edit buffer.

Syntax

[*first*][,*last*]D

where:

first	is the number of the first line to delete.
last	is the number of the last line to delete.

Description

The Delete Lines (D) command removes one or more text lines from the edit buffer. The line after the last line deleted becomes the new current line.

The *first* and *last* line-number parameters define the block of lines to be deleted. (Note that the first line number must be less than or equal to the last line number.) Either or both of these numbers can be omitted (in which case the current line number is used), but a leading comma is required as a placeholder if *first* is omitted when *last* is present. One of the special symbols . (current line) or # (end of buffer) or an expression relative to the current line number (+*n* or −*n*) can be used instead of absolute line numbers.

Examples

If the current line is line 10, to delete the current line, type

```
*10D  <Enter>
```

or

```
*D  <Enter>
```

If the current line is line 10, to delete lines 10 through 15, type

```
*10,15D  <Enter>
```

or

```
*,15D  <Enter>
```

or

```
*,+5D  <Enter>
```

If the current line is line 10, to delete all lines from the current line to the end of the buffer, type

```
*10,#D  <Enter>
```

or

```
*,#D  <Enter>
```

Message

Entry error
The command line contained an error such as a first line number that was greater than the last line number.

EDLIN: E

1.0 and later

End Editing Session

Purpose

Saves the edited file to disk and exits from EDLIN.

Syntax

E

Description

The End Editing Session (E) command writes the contents of the edit buffer to the current directory of the disk in the current drive. If a previously existing file was being edited and there is any text remaining in the original file that has not yet passed through the edit buffer, EDLIN copies this text to the output file. EDLIN gives the newly edited file the same name as the original file and renames the original (unchanged) file with the extension .BAK. Any previous file with the same name and the extension .BAK is lost. EDLIN then returns to MS-DOS.

If the disk does not have enough space to hold the edited file in addition to the original file, EDLIN writes as much of the edited file as possible into a file with the extension .$$$; the remainder of the edited text is lost. The name and contents of the original file are left unchanged.

Example

To end an editing session, type

```
*E  <Enter>
```

Messages

Disk full. Edits lost.
The disk does not contain enough free space for the edited file. A partial file may have been created with the extension .$$$.

File Creation Error
The .BAK file is marked read-only, the root directory is full or cannot contain any more files, or the filename is the same as a volume label or directory name.

No room in directory for file
The file could not be saved because its destination was the root directory and the root directory is full.

Too many files open
MS-DOS was unable to open the .BAK file due to a lack of available system file handles. Increase the value of the FILES command in the CONFIG.SYS file.

EDLIN: I

1.0 and later

Insert Lines

Purpose

Inserts new lines into the edit buffer.

Syntax

[*destination*]I

where:

destination is the number of the line before which text is to be inserted.

Description

The Insert Lines (I) command enables insert mode and allows new text to be placed between previously existing lines of text. When insert mode is terminated, the first line following the inserted lines becomes the new current line.

EDLIN places the new text before the line specified by the *destination* parameter. If *destination* is omitted, EDLIN assumes the current line; if *destination* is larger than the number of lines in the edit buffer, EDLIN simply appends the new text after the actual last line. One of the special symbols . (current line) or # (end of buffer) or an expression relative to the current line number (+*n* or −*n*) can be used instead of an absolute line number.

After an I command, EDLIN issues a prompt consisting of the line number for the inserted text followed by a colon and an asterisk and continues to issue such prompts each time the Enter key is pressed until the user terminates insert mode by pressing Ctrl-C or Ctrl-Break.

Examples

If the current line is line 10, to insert text before line 7, type

```
*7I  <Enter>
```

or

```
*-3I  <Enter>
```

To insert lines at the beginning of the buffer, type

```
*1I  <Enter>
```

To insert lines at the end of the buffer, type

```
*#I  <Enter>
```

Message

Insufficient memory
The edit buffer does not have sufficient room for EDLIN to complete the specified command.

EDLIN: L

1.0 and later

List Lines

Purpose

Displays one or more lines from the edit buffer.

Syntax

[*first*][,*last*]L

where:

first	is the number of the first line to be displayed.
last	is the number of the last line to be displayed.

Description

The List Lines (L) command displays text lines on standard output. If the current line lies within the range of lines listed, EDLIN displays an asterisk next to its number. The current line is not changed.

The *first* and *last* line-number parameters define the block of lines to be listed. (Note that the first line number must be less than or equal to the last line number.) Either or both of these numbers can be omitted, but a leading comma is required as a placeholder if *first* is omitted when *last* is present. One of the special symbols . (current line) and # (end of buffer) or an expression relative to the current line number (+*n* or −*n*) can be used instead of absolute line numbers.

If only the first line number is specified, EDLIN displays text in 23-line increments starting with that number. If only the last line number is specified, EDLIN displays text beginning 11 lines before the current line and continuing to the specified last line. If no line numbers are specified in the command, EDLIN lists the 23 lines centered around the current line; if the current line number is less than 13, EDLIN lists the first 23 lines in the buffer.

Examples

To display lines 20 through 30, type

```
*20,30L  <Enter>
```

If the current line is 20, to display the 23 lines centered around the current line, type

```
*L  <Enter>
```

EDLIN displays lines 9 through 31.

Message

Entry error
The command line contained an error such as a first line number that was greater than the last line number.

EDLIN: M

2.0 and later

Move Lines

Purpose

Moves lines from one place in the edit buffer to another.

Syntax

[*first*],[*last*],*destination*M

where:

first	is the number of the first line to be moved.
last	is the number of the last line to be moved.
destination	is the number of the line before which the moved lines are to be inserted.

Description

The Move Lines (M) command transfers one or more text lines from one location in the edit buffer to another. EDLIN then deletes the original lines and renumbers the edit buffer. The first moved line becomes the new current line.

The *first* and *last* line-number parameters define the block of lines to be moved. (Note that the first line number must be less than or equal to the last line number.) Either or both of these numbers can be omitted (in which case the current line number is used), but the commas must still be entered as placeholders. The *destination* parameter specifies the line before which the moved lines are to be inserted; it is not optional and must not fall within the range of line numbers specified by *first* and *last*. One of the special symbols . (current line) or # (end of buffer) or an expression relative to the current line number (+*n* or −*n*) can be used instead of absolute line numbers.

Example

If the current line is line 10, to move lines 10 through 15 and place them before line 5, type

```
*10,15,5M  <Enter>
```

or

```
*,15,5M  <Enter>
```

or

```
*,+5,-5M  <Enter>
```

Messages

Entry error
The command line contained an error such as a first line number that was greater than the last line number or a destination line number that fell within the range *first,last.*

Must specify destination line number
No destination line number was specified in the command line; therefore, no changes were made to the edit buffer.

EDLIN: P

2.0 and later

Display in Pages

Purpose

Displays lines for viewing in successive screenfuls (pages).

Syntax

[*first*][,*last*]P

where:

first	is the number of the first line to be displayed.
last	is the number of the last line to be displayed.

Description

The Display in Pages (P) command displays text lines on standard output one screenful at a time. Unlike the List Lines (L) command, which has no effect on the current line, P causes the last line displayed to become the new current line. Thus, although the edit buffer is not actually organized into pages, the user can employ repeated P commands to sequentially view successive groups of lines.

The *first* and *last* line-number parameters define the block of lines to be listed; the display starts with the line specified by *first*. (Note that the first line number must be less than or equal to the last line number.) Either or both of these numbers can be omitted, but a leading comma is required as a placeholder if *first* is omitted when *last* is present. If omitted, *first* defaults to the line after the current line and *last* defaults to the line 23 lines after the current line. One of the special symbols . (current line) or # (end of buffer) or an expression relative to the current line number (+*n* or −*n*) can be used instead of absolute line numbers.

Examples

If the current line is 20, to view the next page of lines in the edit buffer, type

```
*P  <Enter>
```

EDLIN displays 23 lines, beginning with line 21, and changes the current line to line 43.

To view successive pages of 23 lines, repeatedly type

```
*P  <Enter>
```

Message

Entry error
The command line contained an error such as a first line number that was greater than the last line number.

EDLIN: Q

1.0 and later

Quit

Purpose

Terminates the editing session without saving the revised file.

Syntax

Q

Description

The Quit (Q) command causes EDLIN to exit without saving any of the changes made to the edited file during the session. The original file's name and contents are left unchanged and no new file is created.

To reduce the danger of accidentally losing the contents of the edit buffer, EDLIN prompts the user for confirmation before carrying out the Q command.

Example

To quit an editing session, type

```
*Q  <Enter>
```

EDLIN issues a prompt for confirmation and, if the response from the user is *Y*, exits to MS-DOS without saving any changes made to the file during the session.

Message

Abort edit (Y/N)?
This prompt is displayed in response to the Q command. Respond with *Y* to exit to MS-DOS without saving changes made to the file; respond with *N* to continue the editing session.

EDLIN: R

1.0 and later

Replace Text

Purpose

Replaces one string in the edit buffer with another.

Syntax

[*first*][,*last*][?]R[*string1*][^Z*string2*]

where:

first	is the number of the first line to be searched.
last	is the number of the last line to be searched.
?	causes the user to be prompted for confirmation before each replacement is made.
string1	is the sequence of characters to be searched for.
^Z	is a Control-Z character.
string2	is the sequence of characters to be substituted for *string1*.

Note: The character limit for the Replace Text command is 127 characters, including both strings and all other parameters.

Description

The Replace Text (R) command substitutes one character string for another within a specified range of lines. The last line in which a replacement occurs becomes the new current line.

The *first* and *last* line-number parameters define the range of lines to be searched for strings to replace. (Note that the first line number must be less than or equal to the last line number.) Either or both of these numbers can be omitted, but a leading comma is required as a placeholder if *first* is omitted when *last* is present. If omitted, *first* defaults to the line after the current line and *last* defaults to the last line in the buffer. One of the special symbols . (current line) or # (end of buffer) or an expression relative to the current line number (+*n* or −*n*) can be used instead of absolute line numbers.

If *string1* is omitted, EDLIN uses the *string1* from the preceding R command; if there was no preceding R command, EDLIN displays an error message. If *string2* is omitted, EDLIN deletes all occurrences of *string1*. *string1* must be separated from *string2* by a Control-Z (^Z) character. If *string1* is omitted, a Control-Z character must still be included to mark the beginning of *string2*, but if *string2* is omitted when *string1* is present, the Control-Z character has no effect and is therefore optional. (The Control-Z character is entered by pressing Ctrl-Z or the F6 key.)

If the ? option is not included in the command line, EDLIN displays each line that contains a match *after* the replacement is carried out. If the ? option is used, EDLIN displays each line containing a match as it is found and prompts the user for confirmation *before* the string is replaced.

The matching operation is case sensitive; EDLIN carries out the substitution only on sequences of characters that match *string1* exactly. Wildcards are not permitted.

Examples

If the current line is line 10, to replace all occurrences of the string *logical* with the string *bitwise* in lines 11 through 20, type

```
*11,20Rlogical^Zbitwise  <Enter>
```

or

```
*,20Rlogical^Zbitwise  <Enter>
```

To cause EDLIN to prompt for confirmation before replacing each string, type

```
*11,20?Rlogical^Zbitwise  <Enter>
```

or

```
*,20?Rlogical^Zbitwise  <Enter>
```

To delete all occurrences of the string *00H* in line 20, type

```
*20,20R00H^Z  <Enter>
```

Messages

Entry error
The command line contained an error such as a first line number that was greater than the last line number.

Insufficient memory
The edit buffer has insufficient room for EDLIN to carry out the specified Replace Text command.

Line too long
The replacement would cause the line being edited to expand beyond 253 characters.

Not found
No occurrence or further occurrences of the string to be replaced were found in the specified range of lines.

O.K.?
If the ? option is used in the command line, this prompt is displayed each time a matching string is found. Respond with *Y* or press the Enter key to replace the string and continue searching; press any other key to leave the string unchanged and continue searching.

EDLIN: S

1.0 and later

Search for Text

Purpose

Searches the edit buffer for a character string.

Syntax

[*first*][,*last*][?]S[*string*]

where:

first	is the number of the first line to be searched.
last	is the number of the last line to be searched.
?	causes the user to be prompted for confirmation before the search is terminated.
string	is the sequence of characters to be searched for (maximum 126 characters).

Description

The Search for Text (S) command searches for a character string within a specified range of lines. When a match is found, EDLIN displays the line containing the match and that line becomes the new current line. If no lines containing the specified string are found, EDLIN displays the message *Not found* and the current line number remains unchanged.

The *first* and *last* line-number parameters define the block of lines to be searched for strings. (Note that the first line number must be less than or equal to the last line number.) Either or both of these numbers can be omitted, but a leading comma is required as a placeholder if *first* is omitted when *last* is present. If omitted, *first* defaults to the line after the current line and *last* defaults to the last line in the buffer. One of the special symbols . (current line) or # (end of buffer) or an expression relative to the current line number (+*n* or −*n*) can be used instead of absolute line numbers.

If *string* is omitted, EDLIN uses the *string* from the last S command or *string1* from the last Replace Text (R) command instead.

If the ? option is not included in the command line, EDLIN displays the first line that contains a match for *string*, makes this the new current line, and terminates the search. If the ? option is used, EDLIN displays each line containing a match for *string* as it is found, followed by an *O.K.?* prompt. If the user responds with *Y* or presses the Enter key, EDLIN terminates the search; if the user presses any other key, the search continues.

The matching operation is case sensitive; EDLIN reports only sequences of characters that match *string* exactly. Wildcards are not permitted.

Examples

If the current line is line 10, to find the first occurrence of the string *xyz* in lines 11 through 20, type

```
*11,20Sxyz  <Enter>
```

or

```
*,20Sxyz  <Enter>
```

To find a particular occurrence of *proc* in the edit buffer, type

```
*1,#?Sproc  <Enter>
```

EDLIN displays the first line containing *proc* and prompts with

```
O.K.?
```

Type *Y* or press Enter to stop the search; press any other key to continue the search.

Messages

Entry error
The command line contained an error such as a first line number that was greater than the last line number.

Not found
No match or no further matches for *string* were found in the specified range of lines.

O.K.?
If the ? option is used in the command line, this prompt is displayed each time a matching string is found. Respond with *Y* or press the Enter key to stop searching; press any other key to continue searching.

EDLIN: T

2.0 and later

Transfer Another File

Purpose

Merges the contents of another file with the file in the edit buffer.

Syntax

[*destination*]T[*drive*:][*path*]*filename*

where:

destination	is the number of the line before which the text from *filename* is to be inserted.
path	is the location of the file to be merged (versions 3.0 and later).
filename	is the name of the disk file from which text is to be merged.

Description

The Transfer Another File (T) command merges the contents of a text file with the current contents of the edit buffer and then renumbers the contents of the edit buffer. The first line of the merged text becomes the current line.

The *destination* parameter specifies the line before which the transferred lines are to be inserted. If omitted, *destination* defaults to the current line. One of the special symbols . (current line) or # (end of buffer) or an expression relative to the current line number (+*n* or −*n*) can be used instead of an absolute line number.

The *filename* parameter specifies the file from which text is to be merged and can include a drive and, in versions 3.0 and later, a path. If a drive or path is not specified, the file to be merged into the edit buffer with the T command *must* be in the current directory of the current drive.

Example

If the current line is line 10, to merge the contents of the file named KEYDEFS.C before line 10 of the edit buffer, type

```
*10Tkeydefs.c  <Enter>
```

or

```
*Tkeydefs.c  <Enter>
```

Messages

File not found
The specified filename does not exist in the current or specified location.

Not enough room to merge the entire file
The space available in the edit buffer is not sufficient to hold the entire file named in the T command. Use the Write Lines to Disk (W) command to partially empty the edit buffer.

EDLIN: W

1.0 and later

Write Lines to Disk

Purpose

Writes lines from the edit buffer to the disk.

Syntax

[*n*]W

where:

n is the number of lines to be written to the file.

Description

If the file being edited is too large to fit into the edit buffer, EDLIN ordinarily reads only enough text to fill 75 percent of the buffer when it opens the file, reserving 25 percent of the buffer for changes and additions to the text. The user must then employ the Write Lines to Disk (W) command and the Append Lines from Disk (A) command to transfer successive blocks of text from the disk until the entire file has passed through the edit buffer. The W command causes EDLIN to write lines to the disk file and delete them from the buffer; then the A command can read new lines from the input file, placing them after the end of the text remaining in the buffer.

The *n* parameter specifies the number of lines to be written to the output file; if *n* is omitted or is larger than the number of lines in the edit buffer, EDLIN writes only enough lines to leave the edit buffer about 25 percent full. EDLIN then renumbers the lines remaining in the edit buffer so that the first remaining line becomes line number one.

Examples

To write 200 lines from the edit buffer to disk (effectively deleting those lines from the buffer), type

```
*200W  <Enter>
```

To write lines from the edit buffer to the disk until the edit buffer is only 25 percent full, type

```
*W  <Enter>
```

EXIT

Terminate Command Processor

2.0 and later

Internal

Purpose

Terminates a secondary copy of the command processor.

Syntax

EXIT

Description

Many communications programs, word processors, database managers, and other application programs load and execute a secondary copy of the system's command processor (COMMAND.COM) to let the user carry out MS-DOS commands without losing the context of the work in progress. Secondary copies of the command processor are also commonly used to execute one batch file under the control of another. (For more information about secondary copies of the command processor, *see* USER COMMANDS: COMMAND.)

The EXIT command cancels a secondary command processor. The terminating processor displays no message and control returns directly to the parent program or command processor.

EXIT has no effect on the currently executing command processor if it was loaded with the /P (permanent) switch or if it is the original command processor (the one loaded during system initialization, when the computer was turned on or restarted).

The EXIT command also allows the user to choose Close from the system menu if a COMMAND window is open under Microsoft Windows.

Example

To terminate the currently executing command processor, type

```
C>EXIT  <Enter>
```

Message

Bad command or filename
The EXIT command did not exist in versions earlier than 2.0, so MS-DOS attempted to execute a nonexistent program named EXIT instead.

FC

2.0 and later

Compare Files

External

Purpose

Compares two files and lists the differences on standard output.

Syntax

FC [/A] [/C] [/L] [/LB*n*] [/N] [*/nnnn*] [/T] [/W] [*drive*:]*pathname1* [*drive*:]*pathname2*

or

FC [/B] [*drive*:]*pathname1* [*drive*:]*pathname2*

where:

pathname1	is the name and location of the first file to be compared, optionally preceded by a drive; wildcard characters are not permitted.
pathname2	is the name and location of the second file to be compared, optionally preceded by a drive; wildcard characters are not permitted.
/A	causes FC to abbreviate the output when comparing ASCII text files (version 3.2).
/B	causes a byte-by-byte (binary) comparison; may not be used with any other switch (default when file extension is .EXE, .COM, .SYS, .OBJ, .LIB, or .BIN).
/C	causes FC to ignore case when comparing alphabetic characters.
/L	causes a line-by-line comparison of two ASCII text files (default when file extension is not .EXE, .COM, .SYS, .OBJ, .LIB, or .BIN) (version 3.2).
/LB*n*	sets the size of the internal line buffer to *n* lines (default = 100) (version 3.2).
/N	includes line numbers on the output of an ASCII file comparison (version 3.2).
/nnnn	is the number of lines that must match to resynchronize during an ASCII file comparison (default = 2; in versions 2.0 through 3.1, range = 1–9, default = 3).
/T	causes FC to compare tabs in text files literally (default = tabs expanded to spaces, with stops at each eighth character position) (version 3.2).
/W	causes FC to ignore spaces, tabs, and blank lines in text files.

Description

The FC utility compares two text files containing lines of ASCII text delimited by new-line characters or two binary files containing data of any type (such as executable programs).

The differences between the two files are listed on standard output, which defaults to the video display but can be redirected to another character device or a file or can be piped to another program.

The FC program first examines the extensions of the two files being compared and, in most cases, selects the appropriate type of comparison automatically. However, the /B switch can be used to force a binary, or byte-by-byte, comparison of the two files named; the /L switch can be used to force a line-by-line comparison. When the /B switch is present, use of the /L, /N, and */nnnn* switches causes an error message to be displayed; any other switches in the command line are ignored.

When comparing ASCII text files, FC loads a buffer with sequential sets of lines from each file and compares the two sets. The size of this buffer defaults to 100 lines but can be modified by including the /LB*n* switch in the command line. If differences are found, the name of the first file, the last matched line, and any mismatched lines from that file are displayed, followed by the first rematched line; then the name of the second file, the last matched line, and any mismatched lines are displayed, followed by the first rematched line from that file. The number of consecutive matching lines that must be detected in order for FC to consider the files resynchronized is controlled with the */nnnn* switch; the default is 2.

If no lines match, if no lines match after the first mismatch, or if the number of mismatched lines exceeds the size of the line buffer, FC displays the message *Resynch failed. Files are too different* (or ****Files are different**** in versions 2.x and 3.0) and terminates.

The /C, /T, and /W switches modify the way in which two text files are compared. The /C switch causes FC to ignore case when comparing alphabetic characters. The /T switch causes FC to compare tab characters (ASCII code 09H) literally, rather than expand them to spaces before comparing corresponding lines. Finally, the /W, or whitespace, switch causes FC to ignore spaces, tabs, and blank lines during the comparison.

The /A and /N switches control the format of the listing of differences between the two text files. The /A switch causes FC to compress the listing of each mismatched set of lines to the first and last lines of each set, separated by ellipsis points. The /N switch causes FC to include the line numbers of the mismatched lines in the display.

During a binary comparison of two files, FC's buffer is reloaded as many times as is necessary to compare the complete files. Unlike the procedure with text-file comparisons, no attempt is made to resynchronize the data if a mismatch is detected and, regardless of the number of mismatches, the comparison process is not terminated. Any differences are displayed with the offset from the start of the file and the actual data from each file. If one file is shorter than the other, FC also displays a warning message at the end of the comparison.

The FC command is present only in MS-DOS. PC-DOS versions 1.0 and later provide a similar function in the COMP command.

Examples

Assume that FILE1.TXT and FILE2.TXT are in the current directory on the disk in the current drive and that they contain the following lines:

FILE1.TXT	FILE2.TXT
First line.	First line.
Second line.	Second line.
Third line.	Third line.
Fourth line.	Fourth line.
Fifth line.	Sixth line.
Sixth line.	Fifth line.
Seventh line.	Seventh line.
Eighth line.	Eighth line.
Ninth line.	Ninth line.
Tenth line.	Tenth line.

To compare these files line by line, type

```
C>FC FILE1.TXT FILE2.TXT  <Enter>
```

This will result in the following display:

```
***** file1.txt
Fourth line.
Fifth line.
Sixth line.
Seventh line.
***** file2.txt
Fourth line.
Sixth line.
Fifth line.
Seventh line.
*****
```

To compare the same two files and produce an abbreviated listing of differences that includes line numbers, type

```
C>FC /A /N FILE1.TXT FILE2.TXT  <Enter>
```

This will result in the following display:

```
***** file1.txt
    4:  Fourth line.
...
    7:  Seventh line.
***** file2.txt
    4:  Fourth line.
...
    7:  Seventh line.
*****
```

Assume that two binary files, FILE1.BIN and FILE2.BIN, are the same length and contain only the following three differences:

Offset	FILE1.BIN	FILE2.BIN
19H	04H	03H
33H	4AH	4BH
42H	52H	51H

To compare these two binary files, type

```
C>FC /B FILE1.BIN FILE2.BIN  <Enter>
```

This will result in the following display:

```
00000019: 04 03
00000033: 4A 4B
00000042: 52 51
```

Note: The use of the /B switch in this example is optional; binary comparison is the default when .BIN files are compared.

Messages

filename* longer than *filename
After all the corresponding data in the two files was compared, data remained in one of the files.

cannot open *filename* - No such file or directory
The specified file cannot be found or does not exist.

DOS 2.0 or later required
FC does not work with versions of MS-DOS earlier than 2.0.

Incompatible switches
The /B switch was used in combination with one or more of the other switches.

Incorrect DOS version
The version of FC is not compatible with the version of MS-DOS that is running.

no differences encountered
The two files being compared are identical.

out of memory
The available memory in the transient program area is insufficient to compare the two files.

Resynch failed. Files are too different
The number of mismatched lines in an ASCII file comparison exceeded the number of lines that can be loaded into FC's comparison buffer (which by default is 100 lines). Rerun the comparison using the /LB*n* switch to allocate a larger buffer.

usage: fc [/a] [/b] [/c] [/l] [/lbNN] [/w] [/t] [/n] [/NNNN] file1 file2
The command line included an invalid switch or FC was entered without any switches or other parameters.

FDISK

3.2

Configure Fixed Disk

External No Net

Purpose

Configures an MS-DOS partition on a fixed disk. This command is included with PC-DOS beginning with version 2.0.

Syntax

FDISK

Description

A fixed disk can be divided into areas of contiguous tracks, or partitions, that are used by different operating systems. A master control record (partition table) on the disk specifies the ID number and the starting and ending disk tracks for each partition. Each fixed disk can have as many as four partitions, but only one partition can be active (bootable) at any given time.

The FDISK utility is a menu-driven program that adds or deletes an MS-DOS partition on a fixed disk, selects one partition as active, and displays the size and status of all partitions. With most implementations of MS-DOS, each fixed disk can contain only *one* MS-DOS partition.

After an MS-DOS partition is created, the FORMAT command must be used to initialize the partition's directory structure. To make it possible to start the computer from the MS-DOS partition on the fixed-disk drive, the /S switch must be used with FORMAT to transfer the operating-system files and the MS-DOS partition must be the active partition.

Warning: If the MS-DOS partition is deleted, any files stored in the partition are irretrievably lost.

Examples

To display the current partitioning of the fixed disk, type

```
C>FDISK  <Enter>
```

The FDISK utility then displays the following menu:

```
Fixed Disk Setup Program Version 0.02
(C) Copyright Microsoft, 1985.

FDISK Options

Choose one of the following:

    1.  Create DOS Partition
    2.  Change Active Partition

    3.  Delete DOS Partition
    4.  Display Partition Data

Enter choice:[1]

Press ESC to return to DOS
```

Note: A fifth option, *Select Next Fixed Drive*, will appear if more than one fixed disk is installed in the system.

Choose option 4 (*Display Partition Data*). FDISK then displays the partition data for the disk in the following form:

```
Display Partition Information

Partition Status  Type   Start  End  Size
    1       A      DOS       0  613   614
Total disk space is 614 cylinders.

Press ESC to return to FDISK Options
```

Assume that the low-level (hardware) formatting for fixed-disk drive C has just been completed by using the drive manufacturer's setup utility. To establish a bootable MS-DOS partition on the disk, type

```
A>FDISK  <Enter>
```

When the menu is displayed, press Enter to choose option 1 (*Create DOS Partition*). FDISK responds with the following message:

```
Create DOS Partition

Do you wish to use the entire fixed
disk for DOS (Y/N).................?[Y]

Press ESC to return to FDISK Options
```

To partition the entire fixed disk for MS-DOS, press Enter to select *Y* (the default). When the FDISK main menu is again displayed, choose option 4 (*Display Partition Data*) to verify that the MS-DOS partition has in fact been established on the fixed disk.

Messages

***n* is not a choice. Please enter Y or N.**
The response to an FDISK prompt requiring a yes or no answer was not *Y* or *N*.

***n* is not a choice. Please enter a choice**
The response to an FDISK prompt requiring a number was not in the proper range or was not a number.

DOS partition created
A new MS-DOS partition has been established on the fixed disk. Use the FORMAT utility to create a directory structure in that partition.

DOS partition deleted
The previously existing MS-DOS partition on the fixed disk has been deleted. Any files contained in the partition are irretrievably lost.

DOS 2.0 or later required
FDISK does not work with versions of MS-DOS earlier than 2.0.

Do you wish to use the entire fixed disk for DOS (Y/N).....?[Y]
Option 1, *Create DOS Partition*, has been chosen from the main menu. Respond with *Y* or press Enter to use all available cylinders for a single DOS partition; respond with *N* to specify that only part of the fixed disk should be used.

Enter starting cylinder number..:[*n*]
Option 1, *Create DOS Partition*, has been chosen from the main menu and the user has responded *N* to the *Do you wish to use the entire fixed disk for DOS?* prompt. This message then prompts for the starting cylinder number of the DOS partition being created.

Enter the number of the partition you want to make active.....:[*n*]
Option 2, *Change Active Partition*, has been chosen from the main menu and this message prompts the user to enter the number of the partition that will become the active partition.

Error loading operating system
An error occurred while attempting to start the system from the fixed disk. Attempt to restart the system. If that fails, start the system from a floppy disk and use the SYS command to copy a new set of the operating-system files to the fixed disk.

Error reading fixed disk
An unrecoverable hardware error was encountered while FDISK was reading data from the fixed disk. The disk may require a low-level (hardware) formatting operation before FDISK can be used; this is usually performed with a special utility program provided by the drive manufacturer.

Error writing fixed disk
An unrecoverable hardware error was encountered while FDISK was writing the new partition control record to the fixed disk. Test the fixed disk with hardware diagnostics before further use.

Fixed disk already has a DOS partition.
The specified fixed disk already contains an MS-DOS partition. Be sure that the correct fixed disk has been selected before proceeding.

Incorrect DOS version
The version of FDISK is not compatible with the version of MS-DOS that is running.

Invalid partition table
The fixed disk's partition table is invalid and the operating system could not be loaded from the fixed disk during system initialization. Restart the computer using a floppy disk and rerun FDISK to determine and correct the problem.

Missing operating system
The DOS partition is the active partition, but it does not contain the operating system. (This message occurs only during system startup.) Use the SYS command to install the operating system.

No DOS partition to delete.
The fixed disk does not contain an MS-DOS partition.

No fixed disks present
FDISK cannot detect a fixed disk in the system. This may reflect a hardware problem with the fixed disk or its controller.

No partitions defined.
This informational message is displayed after the user has chosen option 4, *Display Partition Data*, to indicate that no partitions are currently defined.

No partitions to make active
The fixed disk has not been previously partitioned using FDISK; therefore, an active partition cannot be selected.

No space for a *nnn* cylinder partition.
The fixed disk does not have enough free cylinders to create the desired partition.

No space to create a DOS partition.
The fixed disk does not have enough free cylinders to create an MS-DOS partition.

Partition *n* is already active
The selected partition is already active (bootable); therefore, no action was taken.

Partition *n* made active
This informational message indicates that the selected partition has been made the active partition.

System will now restart
Insert DOS diskette in drive A:
Press any key when ready...
The DOS partition has successfully been created. Strike any key and the system will restart from the disk in drive A.

The current active partition is *n*.
This informational message indicates which partition is currently bootable.

The table partition can't be made active.
The master partition record cannot be made bootable.

Total disk space is *nnn* cylinders.
This informational message indicates the total number of cylinders on the fixed disk.

Total disk space is *nnn* cylinders.
Maximum available space is *nnn*
cylinders at *n*.
The user has responded *N* to the *Do you wish to use the entire fixed disk for DOS?* prompt and this informational message indicates how much space is available for the DOS partition.

Warning: Data in the DOS partition
will be lost. Do you wish to
continue.....?[N]
If the MS-DOS partition is deleted, all files within the partition are lost. Be sure that the files are backed up to another disk before proceeding. Respond with *N* to return to the FDISK main menu; respond with *Y* to delete the DOS partition and lose any files within it.

FIND

Find Character String

2.0 and later

External

Purpose

Searches the character stream from a file or from standard input for a string and displays any lines that contain the string on standard output.

Syntax

FIND [/C] [/N] [/V] *"string"* [[*drive*:][*path*]*filename*] [[*drive*:][*path*]*filename* ...]

where:

string	is the character string to be searched for, always enclosed in quotation marks; case is significant.
filename	is the name of the file to be searched, optionally preceded by a drive and/or path; wildcard characters are not permitted.
/C	displays only the count of the lines containing *string*.
/N	includes the relative line number with each line.
/V	displays only those lines that do *not* contain *string*.

Description

The FIND command searches for all occurrences of a specified string in one or more files (or from standard input). Normally, FIND copies each line in which the string is found to standard output, which defaults to the video display but can be redirected to a file or another character device or can be piped to another program.

The string to be searched for must be enclosed in quotation marks. If the search string itself contains sets of quotation marks, each of those sets of quotation marks must be surrounded by an additional set of quotation marks. FIND's string search is case sensitive.

The search string can be followed by the names of one or more source files; these filenames cannot include wildcards. If no filename is supplied, FIND reads lines from standard input; unless input has been redirected from a file or from the output of another program, this means that FIND reads input from the keyboard. (Keyboard input is terminated by pressing Ctrl-Z or F6 followed by Enter.)

The /C switch counts the total number of lines in which the string appears and sends the count, rather than the lines themselves, to standard output. If the /C switch is used with /V, only the total count of lines that do *not* contain the specified search string is displayed. If both /C and /N are included in the same FIND command, the /N is ignored.

The /N switch includes a relative line number with each line sent to standard output. This is especially helpful when the output of FIND is to be used as a guide to editing the files.

The /V switch reverses the action of FIND so that it copies to standard output all lines that do *not* include the specified string.

Examples

To find and display all lines in the files BREAK.ASM, TALK.ASM, and SHELL.ASM that contain the string *es:*, type

```
C>FIND "es:" BREAK.ASM TALK.ASM SHELL.ASM  <Enter>
```

To find and display all lines in the file STORY.TXT that contain the string *he said "no"*, type

```
C>FIND "he said ""no""" STORY.TXT  <Enter>
```

To search the file \SOURCE\MENUMGR.ASM on the current drive and display all lines that do not contain the string *Error*, type

```
C>FIND /V "Error" \SOURCE\MENUMGR.ASM  <Enter>
```

To obtain a listing on the printer of the lines in the file SHELL.ASM in the current directory of the current drive that contain the string *proc*, including line numbers, type

```
C>FIND /N "proc" SHELL.ASM > PRN  <Enter>
```

To search for all lines that contain two strings, pipe the output of one FIND command to be the input of another. For example, to find only those lines in the file MENUMGR.ASM in the current directory of the current drive that contain both the strings *MOV* and *AX*, type

```
C>FIND "MOV" MENUMGR.ASM | FIND "AX"  <Enter>
```

Messages

---------- *filename*
This informational message gives the name of the file that is currently being searched.

FIND: Access denied
The specified file is locked or being accessed by another application.

FIND: File not found *filename*
The specified file does not exist or the path or drive is not correct.

FIND: Invalid number of parameters
The command line did not include a search string.

FIND: Invalid Parameter *option*
The command line included an invalid switch.

FIND: Read error in *filename*
A disk error occurred during processing of the specified file.

FIND: Syntax error
The command line included an invalid search string. The string must be enclosed in quotation marks.

Incorrect DOS version
The version of FIND is not compatible with the version of MS-DOS that is running.

FORMAT

Initialize Disk

1.0 and later

External No Net

Purpose

Prepares a disk for use by initializing the directory and file allocation table (FAT).

Syntax

FORMAT [*drive*:] [/S] (versions 1.x)

or

FORMAT [*drive*:] [/O] [/V] [/S] (versions 2.0–3.1)

or

FORMAT *drive*: [/1] [/4] [/8] [/N:*n*] [/T:*n*] [/V] [/S] (version 3.2)

or

FORMAT *drive*: [/1] [/B] [/N:*n*] [/T:*n*] (version 3.2)

where:

drive	is the location of the disk to be formatted.
/1	formats a single-sided disk in a double-sided disk drive.
/4	formats a standard double-sided, double-density disk (360 KB) on a quad-density disk drive.
/8	formats a disk with 8 sectors per track.
/B	formats a disk with 8 sectors per track and preallocates space for the hidden operating-system files.
/N:*n*	formats a disk with *n* sectors per track.
/O	formats a disk that is compatible with PC-DOS versions 1.x.
/S	creates a system (bootable) disk; for most implementations of FORMAT, this must be the last switch in the command line.
/T:*n*	formats a disk with *n* tracks.
/V	allows a volume label to be assigned to the disk after formatting.

Note: Each OEM determines which switches will be supported by the FORMAT utility included with the versions of MS-DOS sold with its computers.

Description

The FORMAT command effectively erases any existing data on a disk and creates a new root directory and file allocation table. Each sector of the disk is checked for defects and unusable sectors are marked so that they will not be assigned to files.

If the *drive* parameter is not supplied, the current or default drive is formatted. (A drive letter *must* be specified with version 3.2.) With versions 3.0 and later, the FORMAT program displays a warning if the drive to be formatted is a fixed disk and asks for confirmation before continuing.

When the formatting operation is complete, FORMAT displays the total amount of disk space, the number of bytes lost to defective sectors, the space reserved for or occupied by the hidden operating-system files (if the /B or /S switch was used), and the remaining free disk space. If a floppy disk was formatted, FORMAT then prompts the user to select between formatting another disk and returning to MS-DOS.

Normally, the type of disk drive determines the format that is given to a disk. For example, if a disk is formatted in a standard double-sided, double-density drive, the format defaults to double-sided, 40 tracks per side, 9 sectors per track. The version-specific default formats are 9 or 15 sectors per track with versions 3.0 and later, depending on the drive type; 9 sectors per track with versions 2.x; and 8 sectors per track with versions 1.x. The /1, /4, /8, /N:*n*, and /T:*n* switches can be used to override the default format in some cases. (Not all combinations of /N:*n* and /T:*n* are supported on all hardware.)

Note: A disk formatted with the /4 switch might not be reliably read on a single- or double-sided double-density drive.

The /S switch creates a system (bootable) disk that contains a copy of the operating system. After the format operation is complete, the two hidden files IO.SYS and MSDOS.SYS (or IBMBIO.COM and IBMDOS.COM in PC-DOS) and the nonhidden file COMMAND.COM are copied to the newly formatted disk. Most implementations of FORMAT require that the /S switch, if used, be the last switch in the command line.

The /V switch allows a volume label to be assigned to the new disk. After formatting is complete, FORMAT prompts the user for a volume name, which can be as many as 11 characters. (The characters * ? / \ . , ; : + = < > [ ] and tab are not permitted in a volume label.) Volume labels are displayed by the DIR, CHKDSK, TREE, and VOL commands and, with MS-DOS versions 3.1 and later and PC-DOS versions 3.0 and later, can be modified with the LABEL command after the disk has been formatted.

The /O switch causes FORMAT to write an 0E5H byte at the start of each directory entry so that the resulting disk is compatible with MS-DOS and PC-DOS versions 1.x.

The /B switch formats a disk for 8 sectors per track and reserves room on the disk for the operating-system files. The operating system can then be transferred to the disk with the SYS command to make the disk bootable. The /B switch cannot be used in the same FORMAT command line as the /V or /S switch.

Warning: Disks in drives affected by an ASSIGN, JOIN, or SUBST command should not be formatted. Disks cannot be formatted over a network.

Return Codes

0 The FORMAT operation was successful.
3 The program was terminated by entry of a Ctrl-C or Ctrl-Break.
4 The program was terminated because of a fatal system error (any error other than 0, 3, or 5).
5 The program was terminated by an *N* response to the fixed-disk prompt *Proceed with FORMAT (Y/N)?*

Note: Return codes are available with MS-DOS version 3.2.

Examples

To format the disk in drive B, type

```
C>FORMAT B:  <Enter>
```

In response, FORMAT displays the following message:

```
Insert new diskette for drive B:
and strike ENTER when ready
```

With versions earlier than 3.2, FORMAT then displays the message

```
Formatting ...
```

after the Enter key is pressed, to show that the formatting operation is in progress. With version 3.2, FORMAT displays the message

```
Head: n Cylinder: nn
```

instead, to show the progress of the formatting operation. With all versions, FORMAT displays the following messages if the formatting operation is successful:

```
Format complete
    362496 bytes total disk space
    362496 bytes available on disk

Format another (Y/N)?
```

The byte values may vary depending on the drive type or the switches used in the command line. If bad sectors were encountered during the format operation, FORMAT also displays the number of bytes in bad sectors.

Note: The *Format complete* message overwrites the head/cylinder status line but is appended to the *Formatting ...* status line.

To format and assign a volume label to the disk in drive B, type

```
C>FORMAT B: /V  <Enter>
```

After the usual formatting messages, FORMAT prompts as follows:

```
Volume label (11 characters, ENTER for none) ?
```

The user can then enter a volume name of as many as 11 characters (except * ? / ¦ . , ; : + = < > [ ] or tab), followed by a press of the Enter key.

To format the disk in drive B and make it a system (bootable) disk, type

```
C>FORMAT B: /S  <Enter>
```

FORMAT initializes the disk in the usual manner and then copies the two files containing the operating system (IO.SYS and MSDOS.SYS or IBMBIO.COM and IBMDOS.COM) and the file COMMAND.COM onto the disk. When the formatting operation is completed on a 360 KB floppy disk, the following messages appear:

```
Format complete
System transferred

    362496 bytes total disk space
     62464 bytes used by system
    300032 bytes available on disk

Format another (Y/N)?
```

The number of bytes used by the system will vary with the version of MS-DOS in use.

Messages

***n* bytes total disk space**
***n* bytes used by system**
***n* bytes in bad sectors**
***n* bytes available on disk**
When formatting is complete, FORMAT displays this message with information about space available on the disk. The *bytes used by system* line will not appear if the /S switch was not specified; the *bytes in bad sectors* line will not appear if no bad sectors were found.

Attempted write-protect violation
The disk to be formatted is write protected. Remove the write-protect tab and respond with a *Y* to the *Format another (Y/N)?* prompt.

Cannot find System Files
The /S switch was used and FORMAT was unable to find the necessary system files in the default drive or in drive A.

Cannot FORMAT a Network drive
An attempt was made to format a disk in a drive that has been assigned to a network.

Cannot format an ASSIGNed or SUBSTed drive.
An attempt was made to format a disk in a drive affected by an ASSIGN or SUBST command.

Disk unsuitable for system disk
Defective sectors were detected on the tracks where the operating-system files would normally reside on a bootable disk. Such a disk should be used only for data files, if at all.

Drive letter must be specified
A drive letter must be specified when using version 3.2.

Drive not ready
The floppy-disk drive is empty or the drive door is not closed.

Enter current Volume Label for drive *X*:
The specified drive is a fixed disk, so FORMAT prompts the user to enter the current volume label for verification.

Error in IOCTL call
An internal system error occurred when a pre-version-3.2 block-device driver was used with version 3.2 of FORMAT.

Error reading partition table
FORMAT was unable to read the fixed disk's partition table. Use FDISK on the fixed disk and then try the FORMAT command again.

Error writing directory
FORMAT was unable to create a directory on the disk it is attempting to format. The disk is defective.

Error writing FAT
FORMAT was unable to create the FAT on the disk it is attempting to format. The disk is defective.

Error writing partition table
FORMAT was unable to write the fixed disk's partition table. Use FDISK on the fixed disk and then try the FORMAT command again.

Format another (Y/N)?
At the end of a successful formatting operation or after a nonfatal error, this prompt offers the user the opportunity to format another disk using the same switches specified in the original FORMAT command. Respond with *Y* to format another disk; respond with *N* to return to MS-DOS.

Format complete
The formatting operation has ended. This message contains a number of space characters after it and is printed over the top of the head/cylinder status message, effectively erasing it.

Format failure
The formatting operation was not successful. (This message is usually preceded by another message telling the user why the format failed.) This message contains a number of space characters after it and is printed over the top of the head/cylinder status message, effectively erasing it.

Format not supported on drive *X*:
Device parameters that the computer cannot support were specified in the FORMAT command line.

Formatting...
This informational message indicates that the FORMAT operation is in progress (versions 1.0 through 3.1).

Head: *n* Cylinder: *nn*
This informational message indicates the progress of the FORMAT command during the formatting operation (version 3.2).

Incorrect DOS version
The version of FORMAT is not compatible with the version of MS-DOS that is running.

Insert DOS disk in drive *X*:
and strike ENTER when ready
The /S switch was specified in the FORMAT command line and the disk containing the FORMAT command does not also contain the hidden system files.

Insert new diskette for drive *X*:
and strike ENTER when ready
This prompt allows the user to change disks before the FORMAT operation continues.

Insufficient memory for system transfer
The command line included the /S switch, but available RAM is insufficient to hold the system files during the FORMAT operation.

Invalid characters in volume label
Certain characters (* ? / \ | . , ; : + = < > [ ] and tab) are not allowed in a volume name.

Invalid device parameters from device driver
The DEVICE or DRIVPARM device-driver parameters in the CONFIG.SYS file were incorrectly set or the fixed disk specified in the command line was formatted using MS-DOS versions 2.x without first running FDISK. FORMAT displays this message when the number of hidden sectors is not evenly divisible by the number of sectors per track (meaning that the partition does not start on a track boundary).

Invalid drive specification
The drive specified after the FORMAT command is not a valid drive.

Invalid media or Track 0 bad - disk unusable
One of the switches supplied in the command line is not valid for the drive containing the disk to be formatted (for example, the /8 switch for a quad-density floppy disk) or track 0 of the disk being formatted is unusable to the point that FORMAT is unable to create a directory or file allocation table (FAT).

Invalid parameter
One of the switches supplied in the command line is not valid or is not supported by the version of FORMAT being used.

Invalid volume ID
The volume label entered in response to the *Enter current Volume Label for drive X:* prompt was not the same as the current volume label. Use the VOL command to determine the current volume label.

Non-System disk or disk error
Replace and strike any key when ready
The command line contained a /S or /B switch, but the source disk does not contain the operating-system files.

Not a block device
The drive containing the disk to be formatted is not recognized by MS-DOS as a valid block device.

Parameters not compatible
Switches that cannot be used together were specified in the command line.

Parameters not compatible with fixed disk
One of the switches specified in the command line is not compatible with the specified drive.

Parameters not supported
One of the parameters specified in the command line is not supported by the version of FORMAT being used.

Parameters not Supported by Drive
The device driver for the specified drive does not support generic IOCTL function requests.

Re-insert diskette for drive *X*:
This message prompts the user to reinsert the disk being formatted into the specified drive.

System transferred
The system files IO.SYS and MSDOS.SYS (or IBMBIO.COM and IBMDOS.COM in PC-DOS) and the file COMMAND.COM have been successfully transferred to the newly formatted disk.

Too many open files
FORMAT was unable to write the volume label because insufficient system file handles were available. Increase the value of FILES in the CONFIG.SYS file.

Volume label (11 characters, ENTER for none)?
After formatting a disk with the /V option, FORMAT offers the user the opportunity to enter a volume label for the disk.

Unable to write BOOT
The first track of the disk or MS-DOS partition is bad and cannot be made bootable.

WARNING, ALL DATA ON NON-REMOVABLE DISK
DRIVE *X*: WILL BE LOST!
Proceed with Format (Y/N)?
If a fixed disk is specified as the disk to be formatted, FORMAT warns the user and gives the opportunity to cancel the FORMAT command (versions 3.0 and later).

GRAFTABL

3.0 and later

Load Graphics Character Set

External

Purpose

Installs a RAM-resident table of bitmaps that defines the screen appearance of character codes 128 through 255 in graphics mode.

Syntax

GRAFTABL

Description

On IBM PCs and compatibles in graphics display modes, the video-display BIOS routines (Interrupt 10H) display characters by writing bitmapped matrices of dots to the display. The dot pattern of each screen character's matrix is defined by an entry in a table of bitmaps. The table of bitmaps for the regular ASCII characters, coded 0 through 7FH (0–127), is permanently located in ROM and is always available for use by the system's video driver. The GRAFTABL utility contains a similar table of bitmaps for the upper (extended) characters, coded 80H through 0FFH (128–255). The GRAFTABL command loads this table into RAM and places the address of the table in the vector for Interrupt 1FH.

The GRAFTABL command is not needed for the IBM PCjr or for an enhanced graphics adapter; their ROM BIOS already contains tables of bitmaps for the extended character set.

GRAFTABL is a terminate-and-stay-resident (TSR) program; therefore, its installation reduces the amount of RAM available for use by application programs.

The GRAFTABL command can be executed only once after the computer has been turned on or restarted. An attempt to execute it again will result in an informational message stating that the graphics characters are already loaded.

Example

To load the table of bitmaps for characters 80H through 0FFH (128–255) for use in graphics mode, type

```
C>GRAFTABL  <Enter>
```

Messages

DOS 2.0 or later required
GRAFTABL does not work with versions of MS-DOS earlier than 2.0.

Graphics characters already loaded
The GRAFTABL command has already been executed since the system was turned on or restarted.

Graphics characters loaded
The table of bitmaps has been successfully loaded into RAM and the interrupt vector that points to the table has been initialized.

Incorrect DOS version
The version of GRAFTABL is not compatible with the version of MS-DOS that is running.

GRAPHICS

3.2

Load Graphics Screen-Dump Program

External

Purpose

Installs a resident program that can dump screen contents to the printer in graphics mode. This command is also available with PC-DOS versions 2.0 and later.

Syntax

GRAPHICS (PC-DOS 2.x)

or

GRAPHICS [*printer*] [/B] [/R] (PC-DOS 3.0 and above)

or

GRAPHICS [*printer*] [/B] [/C] [/F] [/P *port*] [/R] (MS-DOS 3.2)

where:

printer is the type of printer to be supported, from the following list:

COLOR1	IBM Personal Computer Color Printer with black ribbon
COLOR4	IBM Personal Computer Color Printer with red-green-blue-black (RGB) ribbon
COLOR8	IBM Personal Computer Color Printer with cyan-magenta-yellow-black (CMY) ribbon
COMPACT	IBM Personal Computer Compact Printer
GRAPHICS	IBM Personal Computer Graphics Printer or compatible (the default)

/B prints the background in color; valid only with the COLOR4 and COLOR8 printers.

/C centers the printout on the page.

/F flips (rotates) the printout 90 degrees.

/P *port* specifies which port the printer is attached to (1–3, where 1 = LPT1, 2 = LPT2, and 3 = LPT3).

/R prints the image as it appears on the screen (white characters on a black background) rather than reversed (the default, black characters on a white background).

Description

The default system routine for dumping the screen to the printer (invoked by Shift-PrtSc) cannot interpret the display in graphics modes. The GRAPHICS command loads a more

sophisticated routine that can dump CGA-compatible graphics displays to several models of IBM graphics printers or compatibles.The GRAPHICS command is not compatible with the Hercules monochrome graphics card or with an enhanced graphics adapter in its enhanced display modes.

If the display is in 640 x 200 graphics mode, the screen dump is printed sideways (rotated 90 degrees). A 320 x 200 graphic can be rotated manually by specifying the /F switch in the command line; however, the image will be elongated horizontally. A rotated image is printed along the left side of the page, which is actually the top of the page in terms of image orientation. The /C option can be used to center a rotated 320 x 200 image on the page.

When used with a printer with a black ribbon, GRAPHICS produces screen dumps with as many as four shades of gray to represent the colors. When used with a printer with a color ribbon (type COLOR4 or COLOR8), GRAPHICS prints all the colors except the background color. With printer types COLOR4 and COLOR8, the /B switch can be used to print the background color also.

Ordinarily, the screen image being dumped is reversed from its appearance on the screen; that is, the light areas on the screen are dark on the printed output and vice versa. The /R switch produces a screen dump that is not reversed in this manner.

If the *printer* parameter is not included in the command line, the GRAPHICS program assumes an IBM Personal Computer Graphics Printer or compatible.

If two or more printers are attached to the system, the /P switch can be used to specify which printer GRAPHICS should use.

The GRAPHICS command is a terminate-and-stay-resident (TSR) program; therefore, its installation reduces the amount of RAM available for use by application programs.

Examples

To load the graphics printing program for use with an IBM Personal Computer Graphics Printer or compatible connected to LPT2, type

```
C>GRAPHICS /P 2  <Enter>
```

Note: A tab, a semicolon character (;), or an equal sign (=) can be used between the /P and the port number instead of a space.

To load the graphics printing program for use with the IBM Personal Computer Color Printer with an RGB ribbon and specify that the background color be printed, type

```
C>GRAPHICS COLOR4 /B  <Enter>
```

To load the graphics printing program for use with the IBM Personal Computer Compact Printer and specify that the images be printed sideways and centered on the page, type

```
C>GRAPHICS COMPACT /F /C  <Enter>
```

Messages

DOS 2.0 or later required
GRAPHICS does not work with versions of MS-DOS earlier than 2.0.

Incorrect DOS version
The version of GRAPHICS is not compatible with the version of MS-DOS that is running.

Unrecognized printer
The printer type specified in the command line is invalid or the printer is not supported.

Unrecognized printer port
The port specified with the /P switch is not a number in the range 1 through 3 or an invalid separator character was used.

JOIN

Join Disk to Directory

3.0 and later

External No Net

Purpose

Joins the directory structure of a disk drive to a subdirectory on another drive.

Syntax

JOIN [*drive1*: *drive2*:*path*]

or

JOIN *drive1*: /D

where:

drive1	is the drive whose directory structure will be joined to a subdirectory of another drive.
drive2:*path*	is the drive and directory that will be used to reference files on *drive1*.
/D	cancels the effect of a previous JOIN command on *drive1*.

Description

The JOIN command allows the directory structure of a disk in one drive to be joined, or spliced, into an empty subdirectory of a disk in another drive. After a JOIN, the entire directory structure of the disk in *drive1*, starting at the root, together with all the files that it contains, appears to be the directory structure of the specified subdirectory on the disk in *drive2*; the drive letter for *drive1* is no longer available. If the directory at the end of the path on *drive2* already exists, it must not contain any files; if it does not exist, JOIN will attempt to create it.

The current directory status of *drive1* has no effect on the JOIN operation. Regardless of which directory or subdirectory is active when the JOIN command is entered, the entire directory structure, including the root directory, is joined to the subdirectory on the disk in *drive2*.

The /D switch cancels any previous JOIN command for a specific drive.

If the JOIN command is entered without parameters, it displays a list of all joins currently in effect.

Warning: The JOIN command should not be used on drives affected by a SUBST or ASSIGN command. Similarly, the BACKUP, RESTORE, FORMAT, DISKCOPY, and DISKCOMP commands should not be used on drives affected by the JOIN command. Drives that have been redirected over a network cannot be joined.

Examples

To join drive B to the subdirectory \DRIVEB on drive C, type

```
C>JOIN B: C:\DRIVEB  <Enter>
```

A subsequent JOIN command without parameters displays

```
B: => C:\DRIVEB
```

To then list the files in the root directory of the disk in drive B, type

```
C>DIR C:\DRIVEB  <Enter>
```

To cancel a previous JOIN command affecting drive B, type

```
C>JOIN B: /D  <Enter>
```

Messages

Cannot JOIN a network drive
A drive assigned to a network cannot be joined to another drive.

Directory not empty
A drive cannot be joined to a directory that already contains files.

DOS 2.0 or later required
JOIN does not work with versions of MS-DOS earlier than 2.0.

Incorrect DOS version
The version of JOIN is not compatible with the version of MS-DOS that is running.

Incorrect number of parameters
There were missing, extra, or incorrect parameters in the command line.

Invalid parameter
A drive cannot be joined to the root directory of any drive.

Not enough memory
The available system memory is insufficient for MS-DOS to run the JOIN command.

KEYB*xx*

Define Keyboard

3.2

External

Purpose

Installs a table that defines the translation of keys to the extended character codes, replacing the default table in the ROM BIOS. This command is included with PC-DOS beginning with version 3.0.

Syntax

KEYB*xx*

where:

xx is a code that selects a keyboard configuration:

DV	Dvorak keyboard (MS-DOS only)
FR	French
GR	German
IT	Italian
SP	European Spanish
UK	United Kingdom English

Note: KEYB*xx* is hardware dependent; therefore, implementation of this command may vary for different OEM versions of MS-DOS.

Description

The KEYB*xx* utility configures the keyboard for use with a language other than United States English, making available special characters that are appropriate for the specified country's language and currency. These special characters are represented by the extended character codes (128–255) that correspond to the characters implemented on the OEM's display adapter. (Both the KEYB*xx* and the GRAFTABL commands must be used to make these characters available in graphics modes on a color/graphics adapter.)

After KEYB*xx* is loaded, special accented characters not part of the language in use are also available through the use of dead keys — keys that are pressed and released before the letter key is pressed. The following dead keys are available on a United States English keyboard for an IBM PC, PC/XT, PC/AT, or strict compatible:

Keyboard Program	Dead Key	Resulting Accent
KEYBGR (Germany)	+	`
	=	´
KEYBFR (France)	[	^
	{	¨
KEYBSP (Spain)	[	´
	]	`
	{	¨
	}	^
KEYBUK (United Kingdom)	Not supported	
KEYBIT (Italy)	Not supported	

The dead-key combinations supported are

Keyboard Program	Combinations Supported
Germany	á é É í ó ú à è ì ò ù
France	ä Ä ë ï ö Ö ü Ü ÿ â ê î ô û
Spain	ä Ä ë ï ö Ö ü Ü ÿ á é É í ó ú à è ì ò ù â ê î ô û
United Kingdom	Dead key not supported
Italy	Dead key not supported

On an IBM PC, PC/XT, PC/AT, or strict compatible, the key sequence Ctrl-Alt-F1 can be used at any time to return the keyboard to the default (United States English) configuration; the sequence Ctrl-Alt-F2 then returns the keyboard to the selected configuration.

KEYB*xx* should be loaded only once during an MS-DOS session; the computer should be restarted if KEYB*xx* is loaded for use with a different language.

KEYB*xx* is a terminate-and-stay-resident (TSR) utility and therefore reduces the amount of memory available to transient application programs (by approximately 2 KB). The only way to reclaim this memory is to restart the system.

Example

To configure the keyboard for Germany, type

```
C>KEYBGR  <Enter>
```

Messages

Bad command or filename
The selected keyboard does not exist or the program that configures the keyboard is not present on the disk.

Incorrect DOS version
The version of KEYB*xx* is not compatible with the version of MS-DOS that is running.

LABEL

Modify Volume Label

3.1 and later

External No Net

Purpose

Adds, alters, or deletes a volume label on a disk. This command is included with PC-DOS beginning with version 3.0.

Syntax

LABEL [*drive*:][*label*]

where:

drive	is any valid disk drive.
label	is a name up to 11 characters long.

Description

With MS-DOS versions 2.0 and later, each disk can have a name called a volume label, which is implemented as a special type of entry in the disk's root directory. With MS-DOS versions 2.x, this volume label can be assigned to a disk only at the time the disk is formatted, using the FORMAT command's /V switch. However, with PC-DOS versions 3.0 and later and MS-DOS versions 3.1 and later, the volume label can be added, modified, or deleted at any time using the LABEL command. (A disk's volume label can be displayed with the VOL command; the label is also included as part of the output from the CHKDSK, DIR, and TREE commands.)

If a new volume name is included in the LABEL command line, the disk's label is changed immediately. If LABEL is entered alone or with only a drive letter, a message is displayed giving the current volume label of the disk in the specified drive (or the default drive, if no drive letter is given) and prompting the user for a new label. (A volume label can be from 1 to 11 characters; it cannot contain any of the characters * ? / \ | . , ; : + = < > [ ] or tab.) If no new volume name is supplied (the user did not type a volume label before pressing Enter), LABEL prompts the user to indicate whether the previous volume label should be deleted. Existing files on the disk are in no way affected by the LABEL command.

The LABEL command cannot be used on a network drive. With MS-DOS version 3.2, the LABEL command also cannot be used on a disk in a drive that is affected by an ASSIGN or SUBST command.

Examples

To give the volume label PAYROLL to the disk in drive B, type

```
C>LABEL B:PAYROLL  <Enter>
```

Note that LABEL immediately overwrites any existing volume label on drive B with the new name; no warning of an existing volume label is given.

To remove the volume label LEDGER from the disk in drive A, type

```
C>LABEL A:  <Enter>
```

The LABEL command displays

```
Volume in drive A is LEDGER
Volume label (11 characters, ENTER for none)?
```

Press the Enter key to receive the additional prompt

```
Delete current volume label (Y/N)?
```

Then respond with *Y* and Enter to remove the volume label from the disk in drive A.

Messages

Cannot LABEL a Network drive
The disk drive specified in the command line cannot be a network drive.

Cannot LABEL a SUBSTed or ASSIGNed drive
The disk drive specified in the command line is currently affected by a SUBST or ASSIGN command (MS-DOS version 3.2).

Delete current volume label (Y/N)?
No volume label was entered in response to the volume-label prompt and a volume label already exists on the disk. Respond with *Y* to delete the current label; respond with *N* to terminate the command.

Incorrect DOS version
The version of LABEL is not compatible with the version of MS-DOS that is running.

Invalid characters in volume label
The characters * ? / \ | . , ; : + = < > [ ] and tab cannot be part of a volume label.

Invalid drive specification
The drive specified in the command line is not valid or does not exist in the system.

No room in root directory
The root directory of the disk in the designated drive is full and a volume label cannot be added. Delete a file or subdirectory from the root directory to make room for the label.

Too many files open
LABEL was unable to write the volume label because no system file handles were available. Increase the value of FILES in the CONFIG.SYS file.

Volume in drive *X* has no label
Volume label (11 characters, ENTER for none)?

or

Volume in drive *X* is *xxxxxxxxxxx*
Volume label (11 characters, ENTER for none)?
This informational message informs the user of the current volume label and prompts the user to add, change, or delete it.

MKDIR or MD

Make Directory

2.0 and later

Internal

Purpose

Creates a new directory.

Syntax

MKDIR [*drive*:][*path*]*new_directory*

or

MD [*drive*:][*path*]*new_directory*

where:

new_directory	is a valid directory name, optionally preceded by an existing path and/or a disk drive.

Description

The MKDIR command creates a directory, adding a branch to the hierarchical directory structure of the disk. If the name of the new directory is preceded by a path, indicating that the new directory is to be a subdirectory of that path, the specified path must already exist.

If *new_directory* is not preceded by an existing path or a backslash character (\), it is presumed to be relative to the current directory. If *new_directory* is preceded by a backslash alone, the directory created will be a subdirectory of the root directory, regardless of the current directory. The length of the full path (including *new_directory*) must not exceed 63 characters.

Warning: The MKDIR command should not be used to create new directories on drives affected by an ASSIGN, JOIN, or SUBST command.

Examples

To create a directory named SOURCE in the current directory of the disk in the current drive, type

```
C>MKDIR SOURCE  <Enter>
```

or

```
C>MD SOURCE  <Enter>
```

To create a directory named LETTERS in the existing directory named WORD (which is a subdirectory of the root directory) on the disk in drive D, type

```
C>MKDIR D:\WORD\LETTERS  <Enter>
```

or

```
C>MD D:\WORD\LETTERS  <Enter>
```

Messages

Invalid drive specification
The drive specified in the command line is not valid or does not exist in the system.

Invalid number of parameters
The name of the new directory was not included in the MKDIR command line.

Unable to create directory
The specified directory cannot be created. This may be caused by a full disk (if the new directory would cause the current directory to be extended), a full root directory (if the new directory's parent is the root directory), the existence of a file or directory with the same name, or an invalid *new_directory* name.

MODE

Configure Device

3.2

External

Purpose

The MODE command has four distinct uses:

- To reconfigure a printer attached to a parallel port (LPT1, LPT2, or LPT3) for printing at 80 or 132 characters per line, 6 or 8 lines per inch, or both (if the printer supports these features). In this form, MODE can also be used to select a parallel printer other than the one attached to LPT1 for use as the default printer.
- To select another display or reconfigure the current display. Reconfiguration includes changing between 40-column and 80-column display, changing between monochrome and color display, centering the display on the screen, or any combination of these.
- To configure the baud rate, parity, and number of databits and stop bits of a serial communications port (COM1 or COM2) for use with a specific printer, modem, or other serial device.
- To redirect printer output from a parallel port to one of the serial ports, so that the serial port becomes the system's default printer port.

Because the syntax for each of these uses of MODE is different, they are discussed separately on the following pages.

Although each form of the MODE command can be issued at the system prompt, MODE commands are commonly used within the AUTOEXEC.BAT file to automatically perform any necessary reconfiguration each time the system is turned on or restarted.

The MODE command is included with PC-DOS beginning with version 1.0.

Message

Incorrect Version of MODE
The version of MODE is not compatible with the version of MS-DOS that is running.

MODE

3.2

Configure Printer

External

Purpose

Sets characteristics for IBM-compatible printers connected to a parallel printer port (LPT1, LPT2, or LPT3). This form of the MODE command is included with PC-DOS beginning with version 1.0.

Syntax

MODE LPT*n*[:][*cpl*] [,[*lpi*][,P]]

where:

LPT*n*	is the parallel printer port (*n* = 1, 2, or 3).
cpl	is the number of characters per line (80 or 132, default = 80).
lpi	is the number of lines per inch (6 or 8, default = 6).
P	causes continuous retries when the printer is not ready.

Description

This form of the MODE command configures an IBM or compatible printer connected to parallel port *n*. Its effect on other printer types may vary. The command has the side effect of canceling any redirection that was previously applied to the specified port with a Redirect Printing MODE command.

The first parameter, LPT*n*, designates the parallel printer port to be configured (LPT1, LPT2, or LPT3). All the other parameters are optional.

The *cpl* parameter selects between printing 80 characters on a line (the default) and 132 characters on a line. The *lpi* parameter selects between 6 lines per inch (the default) and 8 lines per inch. (Note that the attached printer must be capable of printing 132 characters per line or 8 lines per inch and of understanding IBM-compatible printer-control codes; otherwise, specifying these values will have no effect.)

The last parameter in the command line, P, configures the system to retry output continuously (or until Ctrl-Break is pressed) if the printer is not ready or not on line (interpreted by the computer as a time-out error), rather than display an error message. (Note that if P is used and *lpi* is omitted, the comma preceding *lpi* must be specified.) Use of the P option causes part of the MODE program to become permanently resident in memory. (This option is not available in PC-DOS version 1.0.)

Examples

To configure the printer on the first parallel port to print 132 characters per line, with 8 lines per inch, type

```
C>MODE LPT1:132,8  <Enter>
```

To configure the system to continually send output to the printer on the second parallel port if a time-out error occurs but to leave the other values at their defaults, type

```
C>MODE LPT2:,,P  <Enter>
```

Messages

DOS 2.0 or later required
MODE does not work with versions of MS-DOS earlier than 2.0.

Incorrect DOS version
The version of MODE is not compatible with the version of MS-DOS that is running.

Infinite retry of parallel printer timeout
The P option was included in the command line and the system will continuously retry to send output to the printer attached to the specified port if it is not ready or not on line.

INTERNAL ERROR in MODE application
An internal error occurred in the MODE utility and the requested reconfiguration was not carried out.

Invalid parameters
The command line included an incorrect parallel-port specification or one of the configuration parameters was not correct.

LPT*n*: set for 80
The specified printer has been configured for 80 characters per line.

LPT*n*: set for 132
The specified printer has been configured for 132 characters per line.

Printer error
The configuration command could not be carried out because the printer is turned off, not ready, or not on line.

Printer lines per inch set
The printer has successfully been configured for the specified 6 or 8 lines per inch.

Resident portion of MODE loaded
The P option was specified in the command line and part of the MODE command has become permanently resident in memory, decreasing slightly the amount of memory available to other programs.

MODE

3.2

Set Display Mode

External

Purpose

Selects the active video adapter and its display mode or reconfigures the current display. This form of the MODE command is included with PC-DOS beginning with version 2.0.

Syntax

MODE *display*

or

MODE [*display*],*shift*[,T]

where:

display is a video adapter and display mode from the following list:

40	Color/graphics adapter, 40 characters per line
80	Color/graphics adapter, 80 characters per line
BW40	Color/graphics adapter, 40 characters per line, color disabled from composite output
BW80	Color/graphics adapter, 80 characters per line, color disabled from composite output
CO40	Color/graphics adapter, 40 characters per line, color enabled
CO80	Color/graphics adapter, 80 characters per line, color enabled
MONO	Monochrome adapter

shift is R or L, to shift the display left or right one (40-column display) or two (80-column display) character positions.

T causes a test pattern to be displayed for screen alignment.

Description

This form of the MODE command has two uses. The first is to select the active video adapter and its display mode (if more than one adapter is present in the system) or to reconfigure the current adapter. The second is to shift the screen display to the left or right to center it. In both cases, the screen is cleared as a side effect of the command.

The *display* parameter selects the active video adapter and mode or reconfigures the current adapter. If a display adapter that is not available is specified, MODE displays an error message.

The *shift* parameter is simply the single character R or L preceded by a comma. Each shift command causes the screen image to be shifted by two characters if the display adapter is in 80-column mode or by one character if it is in 40-column mode. When the T option is

also included in the command line, the screen image is shifted, a test pattern is displayed, and the user is prompted to indicate whether the screen should be shifted again. Note that use of *shift* causes part of the MODE program to become permanently resident in memory.

Examples

In a system with both a color/graphics adapter and a monochrome display adapter, to select the monochrome display as the active display, type

```
C>MODE MONO  <Enter>
```

To select a color 80-column text mode on the color/graphics adapter, shift the screen image two characters to the left, and display a test pattern, type

```
C>MODE CO80,L,T  <Enter>
```

Messages

DOS 2.0 or later required
MODE does not work with versions of MS-DOS earlier than 2.0.

Do you see the leftmost 0? (Y/N)

or

Do you see the rightmost 9? (Y/N)
When the *shift* and T options are used together, this message allows the user to shift the test-pattern display successive positions until it is properly centered.

Incorrect DOS version
The version of MODE is not compatible with the version of MS-DOS that is running.

INTERNAL ERROR in MODE application
An internal error occurred in the MODE utility and the requested reconfiguration was not carried out.

Invalid parameter
The specified display adapter or mode is not available.

Requested Screen Shift out of range
The display cannot be shifted any further.

Unable to shift Screen left
The screen has already been shifted as far left as possible or the active display adapter cannot be shifted (monochrome or enhanced graphics adapter).

Unable to shift Screen right
The screen has already been shifted as far right as possible or the active display adapter cannot be shifted (monochrome or enhanced graphics adapter).

MODE

3.2

Configure Serial Port

External

Purpose

Controls the configuration of the serial communications adapter. This form of the MODE command is included with PC-DOS beginning with version 1.1.

Syntax

MODE COM*n*[:]*baud*[,*parity*[,*databits*[,*stopbits*[,P]]]]

where:

COM*n*	is the serial port (*n* = 1 or 2).
baud	is the baud rate (110, 150, 300, 1200, 2400, 4800, or 9600).
parity	is the type of parity checking (N = none, O = odd, E = even, default = E).
databits	is the number of bits per character (7 or 8, default = 7).
stopbits	is the number of stop bits (1 or 2, default = 1, except with 110 baud where default = 2).
P	causes continuous retries when the output device is not ready.

Description

This form of the MODE command configures the specified serial port for communication with an external device such as a printer, a terminal, or a modem.

The first parameter, COM*n*, designates the serial port to be configured (COM1 or COM2). Except for the port number and the baud rate, which are required, a parameter can be left unchanged by entering a comma without a value in its position in the command line. (If *all* optional parameters are to be left unchanged and P is not used in the command line, no commas are required.)

The baud rate must be one of the values 110, 150, 300, 600, 1200, 2400, 4800, or 9600. The first two digits can be used as an abbreviation for the full value.

The *parity* parameter specifies the type of parity checking to be done on each character and must be one of the characters N, O, or E (for none, odd, or even, respectively); the default is even parity. The *databits* parameter specifies the length of a character and must be either 7 or 8; the default is 7. The *stopbits* parameter is either 1 or 2. If *baud* is set for 110, the default number of *stopbits* is 2; otherwise, the default is 1.

The last parameter in the command line, P, configures the system to retry output continuously (or until Ctrl-Break is pressed) if the device interfaced to the serial port is not ready or not on line, rather than display an error message. Use of the P option causes part of the MODE program to become permanently resident in memory.

Consult the user's manual for the specific printer, modem, terminal, or other device to determine the proper settings for the MODE parameters.

If a serial printer is to be used instead of LPT1 as the system's default printer, the Redirect Printing MODE command must be specified *after* the Configure Serial Port MODE command.

Example

To configure the first serial port for 9600 baud, no parity, 8 databits, and 1 stop bit, type

```
C>MODE COM1:9600,N,8,1  <Enter>
```

Messages

COM*n: baud, parity, databits, stopbits, timeout*
After the serial port is configured successfully, MODE displays an advisory message confirming the settings. If the P option was not used in the command line, a hyphen character (-) is displayed for *timeout*, to indicate no continuous retries if the printer is not ready or is not on line.

COM port does not exist
The serial port specified in the command line does not exist in the system.

DOS 2.0 or later required
MODE does not work with versions of MS-DOS earlier than 2.0.

Incorrect DOS version
The version of MODE is not compatible with the version of MS-DOS that is running.

INTERNAL ERROR in MODE application
An internal error occurred in the MODE utility and the requested reconfiguration was not carried out.

Invalid baud rate specified
The baud rate included in the command line was not one of the allowed values or was abbreviated incorrectly.

Invalid parameters
The command line specified a COM port that does not exist in the system or one of the configuration parameters for the COM port was not valid.

No COM: ports
The computer does not have any serial ports installed.

Resident portion of MODE loaded
The P option was specified in the command line and part of the MODE command has become permanently resident in memory, decreasing slightly the amount of memory available to other programs.

MODE

3.2

Redirect Printing

External

Purpose

Redirects output from a parallel port to a serial communications port. This form of the MODE command is included with PC-DOS beginning with version 1.1.

Syntax

MODE LPT*n*[:][=COM*n*[:]]

where:

LPT*n*	is the parallel port to be redirected (*n* = 1, 2, or 3).
COM*n*	is the serial port (*n* = 1 or 2) to be used for output instead of LPT*n*.

Description

This form of the MODE command redirects any output for the specified parallel port, sending it to the specified serial communications port instead. The parallel port can be LPT1, LPT2, or LPT3; the serial port can be either COM1 or COM2. A Configure Serial Port MODE command is required *before* the Redirect Printing MODE command, to configure the serial port for the proper baud rate, parity, word length, and stop bits.

Redirection can be canceled by entering *MODE LPT*n alone.

Use of MODE to redirect printer output causes part of the MODE program to become permanently resident in memory. Canceling the redirection will not remove this resident portion from memory.

Example

To cause all output to the first parallel port (LPT1) to be redirected to the first serial port (COM1), type

```
C>MODE LPT1:=COM1:  <Enter>
```

Messages

DOS 2.0 or later required
MODE does not work with versions of MS-DOS earlier than 2.0.

Illegal device name
Either the parallel port or the serial port specified in the command line does not exist in the system.

Incorrect DOS version
The version of MODE is not compatible with the version of MS-DOS that is running.

INTERNAL ERROR in MODE application
An internal error occurred in the MODE utility and the requested reconfiguration was not carried out.

LPT*n*: not redirected
No serial port was specified and any previous redirection from the specified parallel port was canceled.

LPT*n*: redirected to COM*n*:
The MODE command has successfully redirected the output for the specified parallel port to the specified serial port.

Resident portion of MODE loaded
Part of the MODE command has become permanently resident in memory, decreasing slightly the amount of memory available to other programs.

MORE

2.0 and later

Display by Screenful

External

Purpose

Displays output one screenful at a time on standard output.

Syntax

MORE

Description

The MORE filter reads lines of text from standard input and sends them to standard output one screenful (23 lines) at a time. At the end of each screenful, MORE displays the message -- *More* -- and then waits for any key to be pressed before it continues. (Pressing Crtl-C or Ctrl-Break terminates the MORE filter.)

The default input device is the keyboard; the default output device is the video display. Because standard input can be redirected, the MORE filter can also accept input from another character device or a file or from the piped output of another program or filter. Similarly, the output of MORE can be redirected to any character device or file or can be piped to another program (however, the message -- *More* -- will be included with the redirected or piped output).

Examples

To display the file SHELL.C one screenful at a time, type

```
C>MORE < SHELL.C  <Enter>
```

To display the directory of \MASM\SOURCE in the current drive one screenful at a time, pipe the output of the DIR command to the MORE filter by typing

```
C>DIR \MASM\SOURCE | MORE  <Enter>
```

Messages

-- More --
This informational message is displayed at the end of each screenful of text. Press any key to resume output.

MORE: Incorrect DOS version
The version of MORE is not compatible with the version of MS-DOS that is running.

PATH

Define Command Search Path

2.0 and later

Internal

Purpose

Specifies one or more additional drives and/or directories to be searched for a program or batch file if the file cannot be found in the current or specified drive and directory.

Syntax

PATH [*drive*:][*path*][;[*drive*:][*path*]...]

or

PATH ;

where:

drive	is the drive containing the disk to be searched for the executable file.
path	is the name of the directory to be searched for the executable file.

Description

When a command line is entered at the MS-DOS system prompt, the command processor first checks to see if the specified command is one of its internal commands. If it is not, the command processor searches the current directory of the current drive for a file with the same name and the extension .COM, .EXE, or .BAT, in that order. If found, the file is loaded into memory and executed (if the extension is .COM or .EXE) or interpreted by the resident batch-file processor (if the extension is .BAT); otherwise, MS-DOS displays the message *Bad command or file name*, followed by the system prompt. In versions 3.0 and later, a path can precede the command name, causing MS-DOS to make the initial search for a program or batch file under the specified path.

The PATH command designates one or more disk drives and/or directory paths to be searched sequentially for a program or batch file if the file cannot be found in the current or specified drive and directory. The drives and/or directory paths are searched in the order they appear in the PATH command. Multiple *drive*:*path* pairs can be specified, separated by semicolons. A copy of the PATH string is passed to each executing process as a part of the process's environment.

If the *drive* parameter is specified without an associated path, MS-DOS assumes the root directory of *drive*. If the PATH command is followed only by a semicolon, MS-DOS deletes the existing path. If the PATH command is entered with no parameters, MS-DOS displays the existing path.

Invalid or nonexistent drives and/or paths in the PATH command do not result in an error message but are ignored when the PATH string is inspected later during a search for a program or batch file.

The PATH command is generally placed in the AUTOEXEC.BAT file on the system disk so that the search order will be defined each time the system is turned on or restarted.

Examples

To define the directory \BIN on the disk in drive A as the directory to be searched for a program or batch file if the file is not found in the current or specified directory, type

```
C>PATH A:\BIN  <Enter>
```

Subsequent entry of the command

```
C>PATH  <Enter>
```

results in the display

```
PATH=A:\BIN
```

To define the root, \BIN, \DOS, and \DATA directories on drive C and the \UTIL directory on the disk in drive B as the locations to be searched for a program or batch file if the file is not found in the current or specified directory, type

```
C>PATH C:\;C:\BIN;C:\DOS;C:\DATA;B:\UTIL  <Enter>
```

To delete the current search path, type

```
C>PATH ;  <Enter>
```

Message

No Path
The PATH command was entered without parameters and no search path is currently in effect.

PRINT

Print Spooler

2.0 and later

External

Purpose

Loads and configures the background print spooler or adds or deletes files from the print spooler's queue.

Syntax

PRINT [/D:*device*] [/B:*n*] [/M:*n*] [/Q:*n*] [/S:*n*] [/U:*n*] [[*drive*:][*path*]*filename*] [/C][/P] [[[*drive*:][*path*]*filename*] [/C][/P]...]

or

PRINT /T

where:

filename	is the name of the file to be added to or deleted from the print queue, optionally preceded by a drive (and a path with versions 3.0 and later); wildcard characters are permitted.
/B:*n*	sets the print-buffer size in bytes (1–32767, default = 512) (versions 3.0 and later).
/C	deletes the immediately preceding file and all subsequent files from the print queue (until a /P switch is encountered).
/D:*device*	is the character device to be used for printing (default = PRN); must be the first switch, if used (versions 3.0 and later).
/M:*n*	is the length of time in timer ticks that PRINT keeps control during each of its time slices (1–255, default = 2) (versions 3.0 and later).
/P	adds the immediately preceding file and all subsequent files to the print queue (until a /C switch is encountered).
/Q:*n*	is the maximum number of files allowed in the print queue (1–32, default = 10) (versions 3.0 and later).
/S:*n*	is the number of time slices per second that PRINT gives control to the foreground process (1–255, default = 8) (versions 3.0 and later).
/T	terminates printing and empties the print queue.
/U:*n*	is the number of timer ticks that PRINT waits for a busy or unavailable printer or for a disk access or MS-DOS function call to terminate before giving up the time slice (1–255, default = 1) (versions 3.0 and later).

Description

The PRINT utility is a terminate-and-stay-resident (TSR) program that can print files from disk while other programs are running. PRINT maintains a first-in, first-out (FIFO) queue that can hold the names of as many as 32 files. PRINT does not attempt to interpret the contents of a file, except to expand tab characters (ASCII code 09H) with spaces to the next eight-column boundary and to interpret 1AH characters as end-of-file marks. (A program such as PRINT that can transfer files to a printer without any special knowledge of their contents or origin is called a print spooler.)

Note: The PRINT utility continues printing a file until it encounters an end-of-file character (1AH). Therefore, if PRINT is used with nontext files, it may encounter a 1AH character before reaching the end of the file and terminate printing before the entire file has been processed. In such cases, files should be printed using the COPY command, with PRN as the destination.

The PRINT program employs a technique called time-slicing, which is based on its use of the timer-tick interrupt and its detailed knowledge of MS-DOS. PRINT uses this interrupt, which occurs 18.2 times per second on IBM PC-compatible machines, to divide the processor's time between an application or utility program (such as a word processor or a spreadsheet) and the print spooler. Because the application program typically controls the display screen and the keyboard and receives most of the CPU time, it is called the foreground program. The print spooler, which receives a lesser part of the CPU time and usually operates without indicating its status or progress to the operator, is called the background program.

The /B:*n*, /D:*device*, /Q:*n*, /M:*n*, /S:*n*, and /U:*n* switches configure the PRINT utility. These switches are used only the first time the PRINT command is entered after the system has been turned on or restarted.

The /D:*device* switch, which must be the first switch in the command line if used, specifies the peripheral device the print spooler is to use for output. This can be any legal character-output device that is present in the system. If /D:*device* is not included in the first PRINT command, PRINT prompts the user to select an output device (default = PRN). Once an output device has been assigned, a new device cannot be selected without restarting the system.

The /B:*n* switch sets the size of PRINT's file buffer, which controls the amount of data that is read from a file at one time for printing. The value of *n* must be between 1 and 32767 bytes (default value = 512). Large file buffers reduce the amount of extra disk activity caused by the print spooler, but they also reduce the amount of memory available for use by other programs. The /Q:*n* switch controls the size of PRINT's queue—that is, the number of files that can be held in the buffer pending printing. The queue can be configured to hold 1 to 32 files (default = 10).

The /S:*n*, /M:*n*, and /U:*n* switches, available only with versions 3.0 and later, control the time-slicing behavior of PRINT. The /S:*n* switch sets the number of time slices per second—that is, how many times per second—PRINT will be given control; *n* is in the

range 1 through 255 (default = 8). The /M:*n* switch sets the length of time (in timer ticks) that PRINT will keep control during each of its time slices; *n* is in the range 1 through 255 (default = 2). The /U:*n* switch specifies how long (in timer ticks) PRINT should wait for a busy or unavailable printer or for a disk access or MS-DOS function call to terminate before giving up its time slice; again, *n* is in the range 1 through 255 (default = 1). Unless there are special circumstances, the default values for these switches will give acceptable performance.

Files are added to the print queue by entering PRINT followed by one or more pathnames. Files are printed in the order they are placed in the queue. At the end of each file, the print spooler advances the paper to the top of the next page. If a filename containing wildcards is used, all matching files are added to the queue in the order in which they appear in the directory. After a file is queued for printing, it should not be renamed or erased, nor should the disk containing the file be removed, until the printing is complete.

Note: Each print queue entry can be a maximum of 63 characters, including the drive and path.

The /P and /C switches allow files to be added to and deleted from the print queue in the same command line. The /P switch (the default) adds to the print queue the immediately preceding file in the command line and all subsequent files until a /C switch is encountered. Conversely, the /C switch cancels printing for the immediately preceding file in the command line and for all subsequent files until a /P switch is encountered. If a canceled file is currently being printed, PRINT prints the message *File* filename *canceled by operator* on the listing, sounds the printer's alarm (if it has one), and advances the paper to the top of the next page.

The /T switch terminates printing by deleting all files from the print queue. If a file is currently being printed, PRINT prints the message *All files canceled by operator* on the listing, sounds the printer's alarm (if it has one), and advances the paper to the top of the next page.

If PRINT encounters a disk error while attempting to print a particular file, it cancels that file, prints an error message on the printer, sounds the printer's alarm (if it has one), advances the paper to the top of the next page, and goes to the next file in the print queue.

If the PRINT command is entered with no parameters, the contents of the print queue are displayed.

Because PRINT is a TSR utility, it reduces the amount of memory available for use by other programs. The only way to recover the memory occupied by PRINT, even after printing is complete, is to restart the system.

Examples

To install and configure the PRINT program and specify the auxiliary device (AUX) as the printing device, with a print queue that can hold as many as 32 filenames and with a buffer size of 2048 bytes, type

```
C>PRINT /D:AUX /Q:32 /B:2048  <Enter>
```

To add the file DOC.TXT in the current directory of the current drive to the print spooler's queue, type

```
C>PRINT DOC.TXT  <Enter>
```

To delete the file READY.TXT from the print queue and simultaneously add the files FINAL.TXT and REPORT.TXT to the queue, type

```
C>PRINT READY.TXT /C FINAL.TXT /P REPORT.TXT  <Enter>
```

To cancel the file being printed and remove all pending files from the print queue, type

```
C>PRINT /T  <Enter>
```

Messages

***filename* File not found**
A disk was changed or the file was renamed or erased after the PRINT command was entered but before the file was actually printed.

***filename* File not in print queue**
A command line with a /C switch specified a file that is not in the print queue.

***filename* is currently being printed**
This informational message shows which file PRINT is currently printing.

***filename* is in queue**
This informational message shows which file is in the queue waiting to be printed.

***filename* Pathname too long**
The pathname of a file to be printed exceeded 63 characters.

Access denied
An attempt was made to print a locked file.

All files canceled by operator
The /T switch was included in the command line. PRINT terminates printing of the current file, empties the print queue, sounds the printer alarm (if it has one), and advances the paper to the top of the next page.

Cannot use PRINT - Use NET PRINT
If network support has been installed, the NET PRINT command must be used to print files.

Errors on list device indicate that it
may be off-line. Please check it.
The printer has been turned off or placed off line while files are still in the print queue.

File *filename* canceled by operator
A PRINT command was entered with the /C switch to cancel a specific file. If the specified file is currently being printed, PRINT terminates printing of the file, sounds the printer alarm (if it has one), advances the paper to the top of the next page, and resumes printing with the next file in the queue.

Incorrect DOS version
The version of PRINT is not compatible with the version of MS-DOS that is running.

Invalid drive specification
A drive letter specified in the command line is invalid or does not exist in the system.

Invalid parameter
The command line included an invalid switch or configuration switches were used after the first time the PRINT command was used.

List output is not assigned to a device
An invalid destination device was previously entered. Restart the system and specify a valid device in the PRINT command.

Name of list device [PRN]:
This message is displayed in response to the first PRINT command line if the /D:*device* switch was not included. Specify any valid character-output device (default = PRN).

No paper error writing device *device*
An out-of-paper device error was detected while printing on the specified device.

PRINT queue is empty
No files are waiting to be printed.

PRINT queue is full
No additional files can be added to the print queue until the current file is printed. To increase the size of the print queue, restart the system and use the /Q:*n* switch in the PRINT command.

Resident part of PRINT installed
This informational message is displayed on the first entry of a PRINT command to indicate that the PRINT utility is now resident in memory. The amount of memory available to application programs is reduced accordingly.

PROMPT

2.0 and later

Define System Prompt

Internal

Purpose

Defines the form of the command processor's prompt. This command is included in PC-DOS beginning with version 2.1.

Syntax

PROMPT [*string*]

where:

string is a combination of ordinary printable characters and the following special display codes:

Code	Meaning
$b	¦ character
$d	Current date (in the form *Day mm-dd-yyyy*)
$e	Escape character (1BH)
$g	> character
$h	Backspace character (erases the previous character)
$l	< character
$n	Current drive
$p	Current drive and path
$q	= character
$t	Current time (in the form *hh:mm:ss.hh*)
$v	MS-DOS version number
$_	Carriage return/linefeed pair (starts a new line)
$$	$ character

Description

The system's default command processor, COMMAND.COM, displays a prompt on the screen whenever it is ready to accept a command from the user. The command processor determines the format of the prompt from the PROMPT environment variable, if it exists. Otherwise, it uses the default format, which in most OEM implementations of MS-DOS is the letter of the current drive followed by a greater-than sign (for example, C>).

The PROMPT command allows the user to customize the system prompt. This command is usually included in the AUTOEXEC.BAT file so that MS-DOS displays the custom prompt when the system is turned on or restarted.

The *string* parameter can be any combination of printable characters and the special $ control codes listed in the preceding table. The special $ codes allow certain variable information, such as the date and time, to be obtained from the operating system and displayed as part of the prompt. Such system information can be edited in the prompt with the backspace function, which is invoked with the code $h.

Note: When the time is displayed as part of a prompt, it is updated only when the command processor redisplays the prompt.

The escape character, invoked with the code $e, can be used to include standard ANSI escape sequences in *string* to control the appearance of text or its position on the screen. *See* USER COMMANDS: ANSI.SYS for further information on the ANSI escape sequences and the ANSI device driver.

If PROMPT is entered with no parameters, the system prompt is reset to the default format.

The PROMPT command works by modifying the PROMPT environment variable. The same result can be obtained using the SET command with *PROMPT*=string as its argument. *See* USER COMMANDS: SET for further discussion of the environment block and environment variables.

Examples

To define the system prompt as the word *Command* followed by a colon, type

```
C>PROMPT Command:  <Enter>
```

On fixed-disk-based systems it is desirable to display the current drive and path as part of the prompt. To define such a prompt followed by a > character, type

```
C>PROMPT $p$g  <Enter>
```

To define the system prompt to display the time, date, and current drive and path followed by a > character, each on a separate line, type

```
C>PROMPT $t$_$d$_$p$g  <Enter>
```

The system will respond with a display in the following form:

```
16:07:31.56
Thu  6-18-1987
C:\BIN\DOS>
```

To create a prompt that displays the time without the seconds and hundredths of a second, followed by a space and the date without the year, followed by a space and the current drive and a > character, type

```
C>PROMPT $t$h$h$h$h$h $d$h$h$h$h$h $n$g  <Enter>
```

The system will respond with

```
16:07 Thu  6-18 C>
```

To define a prompt that always displays the current time and date in the upper right corner of the screen before displaying the current drive and the > character on the current line, type

```
C>PROMPT $e[s$e[0;60H$t$h$h$h$h$h$h $d$e[u$n$g  <Enter>
```

The escape sequence *$e[s* saves the current cursor position; the sequence *$e[0;60H* positions the cursor at row 0, column 60; the next several codes format the date and time; the sequence *$e[u* restores the original cursor position. (This example requires that the ANSI driver be loaded to interpret the escape sequences.)

RAMDRIVE.SYS

3.2

Virtual Disk

External

Purpose

Creates a virtual disk in memory.

Syntax

DEVICE=[*drive*:][*path*]RAMDRIVE.SYS [*size*] [*sector*] [*directory*] [/A ¦ /E]

where:

size	is the size of the virtual disk in kilobytes (minimum = 16, default = 64).
sector	is the sector size in bytes (128, 256, 512, or 1024; default = 128).
directory	is the maximum number of entries in the virtual disk's root directory (3–1024, default = 64).
/A	causes RAMDRIVE to use Lotus/Intel/Microsoft Expanded Memory for storage (cannot be used with /E).
/E	causes RAMDRIVE to use extended memory for storage (cannot be used with /A).

Note: Unless a /A or /E switch is used, the virtual disk is created in conventional memory.

Description

The RAMDRIVE.SYS installable device driver allows the configuration of one or more virtual disks (sometimes referred to as electronic disks or RAMdisks). A virtual disk is implemented by mapping a disk's structure — directory, file allocation table, and files area — onto an area of random-access memory, rather than onto actual sectors located on a magnetic recording medium. Access to files stored on a virtual disk is very fast, because no moving parts are involved and the "disk" operates at the speed of the system's memory.

Warning: Because a RAMdisk resides entirely in RAM and is therefore volatile, any information stored there is irretrievably lost when the computer loses power or is restarted.

RAMDRIVE.SYS can create a virtual disk in conventional memory, extended memory, or Lotus/Intel/Microsoft Expanded Memory. Conventional memory is the term for the up-to-640 KB of RAM that contain MS-DOS and any application programs. Extended memory is the term for the memory at addresses above 1 MB (100000H) that is available on 80286-based personal computers such as the IBM PC/AT. Expanded memory is the term for a subsystem of bank-switched memory boards (and a driver to manage them) that is compatible with the Lotus/Intel/Microsoft Expanded Memory Specification (LIM EMS).

A virtual disk can be installed in conventional memory by simply inserting the line *DEVICE=RAMDRIVE.SYS* into the system's CONFIG.SYS file and restarting the system. A

new "drive" then becomes available in the system, with a default size of 64 KB, 128-byte sectors, and 64 available directory entries (assuming memory is sufficient). The virtual disk is assigned the next available drive letter (which is displayed in RAMDRIVE's sign-on message). The drive letter assigned depends on the number of other physical and virtual disks in the system and also on the position of the *DEVICE=RAMDRIVE.SYS* line in the CONFIG.SYS file relative to other installed block devices. Available memory permitting, multiple virtual disks can be created by using multiple *DEVICE=RAMDRIVE.SYS* lines. Several optional parameters allow the user to customize the size and configuration of the virtual disk and to use extended memory or expanded memory if it is available.

The *size* parameter specifies the amount of RAM, in kilobytes, to be allocated to the virtual disk. The default is 64 KB, but any size from 16 KB to the total amount of available memory can be specified.

The *sector* parameter sets the virtual sector size used within the virtual disk. The *sector* value can be 128, 256, 512, or 1024 bytes (default = 128 bytes). Selection of the smallest sector size results in a minimum of wasted virtual disk space per file but also results in a somewhat slower transfer of data. Physical disk devices on IBM PC-compatible systems always use 512-byte sectors.

Warning: The 1024-byte sector size is not supported in most implementations of MS-DOS and will terminate the installation of RAMDRIVE.SYS if it is used. Check the documentation included with the computer to see if this value is supported.

The *directory* parameter sets the number of available entries in the virtual disk's root directory. The allowed range is 3 to 1024 (default = 64). Each directory entry requires 32 bytes. RAMDRIVE rounds the number of available directory entries up, if necessary, so that an integral number of sectors are assigned to the root directory.

The /A switch causes Lotus/Intel/Microsoft Expanded Memory to be used for the virtual disk, rather than conventional memory; the /E switch causes extended memory to be used. Either option allows very large virtual disks to be configured while still leaving the maximum amount of conventional memory available for use by application programs. The /A and /E switches cannot be used together.

Note: If RAMDRIVE uses conventional memory for virtual disk storage, the memory cannot be reclaimed except by modifying the CONFIG.SYS file and restarting the system.

Examples

To create a virtual disk drive with the default values of 64 KB disk size, 128-byte sectors, and 64 available directory entries, include the following command

```
DEVICE=RAMDRIVE.SYS
```

in the CONFIG.SYS file and restart the system.

To create a 4 MB virtual disk drive in Lotus/Intel/Microsoft Expanded Memory, with 512-byte sectors and 224 available directory entries, when RAMDRIVE.SYS is located in a directory named \DRIVERS on drive C, include the command

```
DEVICE=C:\DRIVERS\RAMDRIVE.SYS 4096 512 224 /A
```

in the CONFIG.SYS file and restart the system.

Messages

Microsoft RAMDrive version *n.nn* virtual disk *X*:
Disk size: *nn*k
Sector size: *nnn* bytes
Allocation unit: *n* sectors
Directory entries: *nnn*

RAMDRIVE.SYS was successfully installed and this message informs the user of the version of RAMDRIVE.SYS that created the virtual disk, the drive letter assigned to the disk, and the characteristics of the disk.

RAMDrive: Above Board Memory Manager not present
The /A switch was used in the command line and the Lotus/Intel/Microsoft Expanded Memory Manager is not present in the system. Place the DEVICE command that loads the memory manager *before* the *DEVICE=RAMDRIVE.SYS* command in the CONFIG.SYS file.

RAMDrive: Above Board Memory Status shows errors
The Above Board device driver is bad or damaged or the board itself is defective. Consult the Above Board manual or the manufacturer.

RAMDrive: Computer must be PC-AT, or PC-AT compatible.
The /E switch was used in the command line and the computer is not an 80286-based IBM PC/AT or compatible.

RAMDrive: Incorrect DOS version
The version of RAMDRIVE.SYS is not compatible with the version of MS-DOS that is running.

RAMDrive: Insufficient memory
Available memory is insufficient for RAMDRIVE.SYS to create a virtual drive.

RAMDrive: Invalid parameter
One of the parameters supplied in the command line is incorrect or is not supported by the computer.

RAMDrive: I/O error accessing drive memory
The Expanded Memory Manager device driver is bad or damaged or the board itself is defective. Consult the board's manual or contact the manufacturer.

RAMDrive: No extended memory available
The /E switch was specified but the system does not contain extended memory.

RECOVER

2.0 and later

Recover Files

External No Net

Purpose

Reconstructs files from a disk that has developed unreadable sectors or has a damaged directory.

Syntax

RECOVER *drive*:

or

RECOVER [*drive*:][*path*]*filename*

where:

drive	is the letter of the drive holding the disk with a damaged directory.
filename	is the name of the file that will be reconstructed, optionally preceded by a drive and/or path; wildcard characters are not permitted.

Description

The RECOVER command partially rescues a file on a disk that has developed bad sectors by deleting the bad sectors from the file. RECOVER can also reconstruct files (including files stored in subdirectories) from a disk that has a damaged directory.

When RECOVER is used with a filename, the file is read allocation unit by allocation unit; unreadable allocation units are marked as bad and are no longer allocated to the file. The resulting file is usable, although the data contained in the bad allocation units is lost. (The recovered file may or may not be reusable by the specific application that created it.) The directory entry for *filename* is also adjusted to reflect the sectors that were lost and the bad sectors are marked in the disk's file allocation table so that they are not reused for another file.

If a disk's directory is damaged, it still may be possible to recover all the files on the disk and build a new directory by using RECOVER with *drive* as the only command-line parameter. RECOVER completely erases the previous contents of the damaged directory and constructs new directory entries for each of the original files by inspecting the disk's file allocation table. The recovered files receive names of the form FILE*nnnn*.REC, starting with FILE0001.REC. Each recovered file's size is always a multiple of the disk cluster size, so recovered files may require editing to eliminate spurious data at the ends of the files.

RECOVER restores each subdirectory as an individual file that contains the names of the files originally stored in it. The actual files contained within those subdirectories are also reconstructed, although they are no longer associated with the subdirectory in which they

originally resided. Restored files and subdirectories, regardless of their location on the damaged disk, are placed in the new root directory. If there are more files on the damaged disk than can be contained in the new root directory (for example, more than 112 for a 5.25-inch, 360 KB floppy disk), the user must repeat the RECOVER command after copying the already-recovered files to another disk and deleting them from the damaged disk.

Examples

To recover the file MENUMGR.C in the current directory of the current drive, type

```
C>RECOVER MENUMGR.C  <Enter>
```

To recover all files on the disk in drive B, which has a damaged directory, type

```
C>RECOVER B:  <Enter>
```

Messages

***n* file(s) recovered**
When RECOVER is used on a disk with a damaged directory, this informational message is displayed at the conclusion of processing to indicate how many files of the form FILE*nnnn*.REC were constructed.

***n* of *n* bytes recovered**
When RECOVER is used on a damaged file, this informational message is displayed at the conclusion of processing to advise how many bytes of the file were recovered.

Cannot RECOVER a Network drive
Files on a drive assigned to a network cannot be recovered.

File not found
The file specified in the command line cannot be found or does not exist.

Incorrect DOS version
The version of RECOVER is not compatible with the version of MS-DOS that is running.

Invalid drive or file name
An invalid drive letter was specified or the filename contains a wildcard.

Invalid number of parameters
More than one drive letter or filename was specified in the command line.

Press any key to begin recovery of the file(s) on drive *X*
This prompt message gives the user the opportunity to change disks after the RECOVER program is loaded but before processing begins.

Warning - directory full
New directory entries for the reconstructed files cannot be created because the root directory is full. Copy the recovered files to another disk, delete them from the damaged disk, and then repeat the RECOVER command on the damaged disk.

RENAME or REN

1.0 and later

Change Filename

Internal

Purpose

Changes the name of a file or set of files.

Syntax

RENAME [*drive*:][*path*]*oldname newname*

or

REN [*drive*:][*path*]*oldname newname*

where:

oldname	is the name of an existing file or set of files, optionally preceded by a drive and/or path; wildcard characters are permitted.
newname	is the new name to be assigned to *oldname*; wildcard characters are permitted, but a drive and/or path cannot be specified.

Description

The RENAME command changes the name of an existing file or set of files. It does not make copies of files or move files from one location in the disk's directory structure to another or from one drive to another.

The *oldname* parameter can refer to a single file or can include wildcards to specify a set of files; a drive and path can be included as part of *oldname*.

The *newname* parameter specifies the new name to be given to the file or files; it cannot include a drive or path. A wildcard in *newname* causes that portion of the original filename to be left unchanged. If the new name for a file is the same as the name of an existing file, RENAME terminates with an error message.

Examples

To rename the file REVS.DOC, located in the current directory of the current drive, to CHANGES.TXT, type

```
C>RENAME REVS.DOC CHANGES.TXT  <Enter>
```

or

```
C>REN REVS.DOC CHANGES.TXT  <Enter>
```

To rename all files with a .DOC extension in the \SOURCE directory on the disk in drive D to have a .TXT extension, type

```
C>REN D:\SOURCE\*.DOC *.TXT  <Enter>
```

Messages

Duplicate file name or File not found
The new name specified for a file already exists or a file with the old name cannot be found or does not exist.

Invalid directory
The command line included a reference to a directory that is invalid or does not exist.

Invalid drive specification
The command line included a reference to a disk drive that is invalid or does not exist in the system.

Invalid number of parameters
The command line included too few or too many filenames.

Invalid parameter
The *newname* parameter in the command line included a drive and/or path.

REPLACE

3.2

Update Files

External

Purpose

Selectively adds or replaces files on a disk.

Syntax

REPLACE [*drive*:]*pathname* [*drive*:][*path*] [/A][/D][/P][/R][/S][/W]

where:

pathname	is the name and location of the source files to be transferred, optionally preceded by a drive; wildcard characters are permitted in the filename.
drive:path	is the destination for the file being transferred; filenames are not permitted in the destination parameter.
/A	transfers only those source files that do not exist at the destination (cannot be used with /S or /D).
/D	transfers only those source files with a more recent date than their destination counterparts (cannot be used with /A).
/P	prompts the user for confirmation before each file is transferred.
/R	allows REPLACE to overwrite destination read-only files.
/S	searches all subdirectories of the destination directory for a match with the source files (cannot be used with /A).
/W	causes REPLACE to wait for the disk to be changed before transferring files.

Description

The REPLACE utility allows files to be updated easily to more recent versions. REPLACE examines the source and destination directories and, depending on the switches used in the command line, selectively updates matching files or copies only those files that exist on the source disk but not the destination disk.

The *pathname* parameter (the source) specifies the name and location of the files to be transferred (optionally preceded by a drive); wildcards are permitted in the filename. The *drive:path* parameter (the destination) specifies the location of the files to be replaced and can consist of a drive, a path, or both. If only a drive is specified as the destination, REPLACE assumes the current directory of the disk in that drive. If the destination is omitted completely, REPLACE assumes the current drive and directory. The /S switch causes REPLACE to also search all subdirectories of the destination directory for files to be replaced.

The /A, /D, and /P switches allow selective replacement of files on the destination disk. When the /A switch is used, REPLACE transfers only those files on the source disk that do not exist in the destination directory. When the /D switch is used, REPLACE transfers only

those source files that match the destination filenames but have a more recent date than their destination counterparts. (The /D switch is not available with the PC-DOS version of REPLACE.) The /P switch causes REPLACE to prompt the user for confirmation before each file is transferred.

The /R switch allows the replacement of read-only as well as normal files. If the /R switch is not used and one of the destination files that would otherwise be replaced is marked read-only, the REPLACE program terminates with an error message. (REPLACE cannot be used to update hidden or system files.)

The /W switch causes REPLACE to pause and wait for the user to press any key before beginning the transfer of files. This allows the user to change disks in floppy-disk systems with no fixed disk and in those cases where the REPLACE program itself is present on neither the source nor the destination disk.

Return Codes

0	The REPLACE operation was successful.
1	An error was found in the REPLACE command line.
2	No matching files were found to replace.
3	The source or destination path was invalid or does not exist.
5	One of the files to be replaced was marked read-only and the /R switch was not included in the command line.
8	Memory was insufficient to run the REPLACE command.
15	An invalid drive was specified in the command line.
Other	Standard MS-DOS error codes (returned on a failed Interrupt 21H file-function request).

Examples

To replace the files in the directory \SOURCE on the current drive with all matching files on the disk in drive A that have a more recent date, type

```
C>REPLACE A:*.* \SOURCE /D  <Enter>
```

To transfer from the disk in drive A only those files that are not already present in the current directory, type

```
C>REPLACE A:*.* /A  <Enter>
```

Messages

_n_ File(s) added
After the replacement operation is completed, if the /A switch was used in the command line, REPLACE displays the total number of files added.

_n_ File(s) replaced
After the replacement operation is completed, REPLACE displays the total number of files processed.

Access denied '*pathname*'
One of the files to be replaced on the destination disk is marked read-only and the /R switch was not included in the command line.

Add *pathname*? (Y/N)
The /A and /P switches were specified in the command line and REPLACE prompts the user for confirmation before adding each file.

Adding *pathname*
The /A switch was specified in the command line and REPLACE displays the name of each file it adds.

File cannot be copied onto itself '*pathname*'
The source and destination command-line parameters specified the same file in the same location.

Incorrect DOS Version
The version of REPLACE is not compatible with the version of MS-DOS that is running.

Insufficient disk space
The destination disk does not have enough available space to hold the files being added or replaced.

Insufficient memory
The system does not have enough RAM available to process the REPLACE command.

Invalid drive specification '*X*:'
The command line specified a disk drive that is invalid or does not exist in the system.

Invalid parameter '*switch*'
The command line included a switch that is not supported by the REPLACE command.

No files added
The /A switch was used and the specified file(s) already exist on the destination disk.

No files found '*pathname*'
The files to be added or replaced on the destination disk were not found on the source disk.

No files replaced
The files at the destination are identical with the files on the source disk or do not meet the criteria specified by the switches.

Parameters not compatible
The command line included two or more switches that cannot be used together.

Path not Found '*pathname*'
The source or destination parameter included a nonexistent path or directory.

Path too long
The source or destination parameter included a path element that is too large (probably because of a missing backslash character [\]).

Press any key to begin adding file(s)
The /W and /A switches were specified in the command line and REPLACE waits for the user to press a key before proceeding, allowing disks to be changed.

Press any key to begin replacing file(s)
The /W switch was specified in the command line and REPLACE waits for the user to press a key before proceeding, allowing disks to be changed.

Replace *pathname*? (Y/N)
The /P switch was specified in the command line and REPLACE prompts the user for confirmation before replacing the file.

Replacing *pathname*
This informational message indicates the progress of the REPLACE command by displaying the name of each file as it is being replaced.

Source path required
Although the destination parameter can usually be omitted and defaults to the current drive and directory, the source location for the files to be replaced must always be specified.

Unexpected DOS Error *n*
This message usually indicates a bad or damaged disk. Use the CHKDSK command to determine the problem.

RESTORE

2.0 and later

Restore Backup Files

External

Purpose

Restores files from a disk created with the BACKUP command.

Syntax

RESTORE *drive1*: [*drive2*:][*pathname*] [/A:*date*] [/B:*date*] [/E:*time*] [/L:*time*][/M][/N] [/S][/P]

where:

drive1	is the drive that contains the backup files created by the BACKUP command.
drive2	is the drive to which the backup files will be restored.
pathname	is the name of the file(s) to be restored from *drive1*; wildcard characters are permitted in the filename. If a path is used, a filename must be specified.
/A:*date*	restores files that were modified on or after *date*.
/B:*date*	restores files that were modified on or before *date*.
/E:*time*	restores files modified at or before *time*.
/L:*time*	restores files modified at or after *time*.
/M	restores only files modified since the last backup.
/N	restores only files that no longer exist on the destination disk.
/P	prompts the user for confirmation before restoring hidden or read-only files or before overwriting files that have changed since they were last backed up .
/S	restores all files in the subdirectories of the specified directory, in addition to the files in the specified directory.

Note: The PC-DOS version of RESTORE supports only the /P and /S switches.

Description

The RESTORE command restores files from a backup disk or directory created with the BACKUP command to their original location in a directory structure. Before version 3.1, the RESTORE command could restore files only from one floppy disk to another or from a floppy disk to a fixed disk. With later versions, RESTORE can also restore files from one fixed disk to another or from a fixed disk to a floppy disk.

The *drive1* parameter specifies the source for the backed-up files. If the source disk is a fixed disk, the backup files are always obtained from the directory \BACKUP. If multiple floppy disks were used to hold the backed-up files, RESTORE prompts the user for each disk as it is required.

The destination can be any combination of a drive, a path, and a filename; the filename can include wildcards. If the destination drive is omitted, MS-DOS assumes the current drive. If a path is not specified, the files are restored to the current directory. (Note that files must be restored to the same directory they were backed up from.) If a path is specified, a filename must be specified as well. If neither a path nor a filename is included in the command line, all directories, subdirectories, and files on the backup disk(s) are restored to the destination disk. The /S switch can be used to force restoration of the files in all the subdirectories of a named directory.

Files are restored in the order they were backed up, regardless of their current order on the destination disk. If files with the same name and location already exist on the destination disk, they are replaced by the backup copies.

The RESTORE program supports a number of switches that allow selective restoration of files from the backup disk. The /A:*date*, /B:*date*, /E:*time*, and /L:*time* switches allow files to be restored based on the time and/or date they were backed up. The /M switch restores only those files that have been changed on the destination disk since the backup disk was created. The /P switch prompts the user before restoring a hidden or read-only file or a file that has been changed since it was last backed up.

The MS-DOS and PC-DOS RESTORE programs are compatible except when a /A:*date*, /B:*date*, /E:*time*, /L:*time*, /M, or /N switch is used. These switches are not supported in the PC-DOS version.

Warning: The RESTORE command should not be used on a disk drive affected by an ASSIGN, SUBST, or JOIN command.

Return Codes

0 The restore operation was successful.
1 No files were found to restore.
2 Some files were not restored because of a file-sharing conflict (versions 3.0 and later).
3 The restore operation was terminated by the user.
4 The program was terminated by an unrecoverable (critical) hardware error.

Examples

To restore the file named MENUMGR.C from the backup disk in drive A to the directory named \SOURCE on the disk in drive B, type

```
C>RESTORE A: B:\SOURCE\MENUMGR.C  <Enter>
```

To restore all the files on the backup disk in drive A to their original locations in the directory structure of drive C, type

```
C>RESTORE A: C:\*.* /S  <Enter>
```

To restore all the files with the extension .C from the backup disk in drive A to the directory named \SOURCE on drive C, requesting confirmation for those files that are read-only or hidden, type

```
C>RESTORE A: C:\SOURCE\*.C /P  <Enter>
```

Messages

***** Files were backed up at *time* on *date* *****
This informational message shows when the BACKUP command was used on the backed-up files.

***** Not able to restore file *****
The backup file or the destination disk contains an error. Use the CHKDSK command to determine the problem.

***** Restoring files from drive *X*: *****
Diskette: *n*
This informational message indicates the progress of the RESTORE command.

DOS 2.0 or later required
RESTORE does not work with versions of MS-DOS earlier than 2.0.

File creation error
The destination directory is full. This usually occurs only if the destination is the root directory but can also happen if a file is being restored to a subdirectory and the disk itself is full.

Incorrect DOS version
The version of RESTORE is not compatible with the version of MS-DOS that is running.

Insert backup diskette *n* in drive *X*:
Strike any key when ready
This message prompts the user to insert the next backup disk in sequence. Disks used in multidisk backups should always be labeled and numbered during a BACKUP operation.

Insert restore target diskette in drive *X*:
Strike any key when ready
This prompt is displayed when files are being restored to a floppy disk.

Insufficient memory
Available memory is not sufficient for the RESTORE program to execute.

Invalid drive specification
The command line included a drive that is invalid or does not exist in the system.

Invalid number of parameters
The command line included too many or too few parameters.

Invalid parameter
The command line included an invalid switch or other parameter.

Invalid path
The destination parameter included a path that is invalid or does not exist.

Restore file sequence error
Files are being restored from a multidisk set of backup disks and a floppy disk was used out of order.

Source and target drives are the same
Files cannot be restored from a drive to the same drive.

Source does not contain backup files
The files on the backup disk are not in the special format used by the BACKUP and RESTORE programs.

System files restored
Target disk may not be bootable
The backup disk included copies of the hidden operating-system files MSDOS.SYS and IO.SYS (or IBMDOS.COM and IBMBIO.COM in PC-DOS) and these files were restored to the destination disk. The destination disk is bootable only if these two files are the first files on the disk and IO.SYS (or IBMBIO.COM) is written into contiguous clusters.

Target is full
The destination disk is full and no further files can be restored.

Target is Non-Removable
The disk to which files are being restored is not removable.

The last file was not restored
The destination disk is full or the last file on the backup disk was bad.

Warning! Diskette is out of sequence
Replace diskette or continue if okay
Files are being restored from a multidisk set of backup disks and a floppy disk was used out of order.

Warning! File *filename*
is a hidden file
Replace the file (Y/N)?
The backed-up file has the same filename as a hidden file on the destination disk, which may be overwritten. (This message appears only if the /P switch was used.) Respond with *Y* to overwrite the file on the destination disk; respond with *N* to leave the destination file unchanged and continue the RESTORE operation.

Warning! File *filename*
is a read-only file
Replace the file (Y/N)?
The backed-up file has the same name as a read-only file on the destination disk, which may be overwritten. (This message appears only if the /P switch was used.) Respond with

Y to overwrite the file on the destination disk; respond with *N* to leave the destination file unchanged and continue the RESTORE operation.

Warning! File *filename*
was changed after it was backed up
Replace the file (Y/N)?
Data has been changed or added to the destination file since the backup disk was created and this data will be lost if the file is restored. (This message appears only if the /P switch was used.) Respond with *Y* to restore the backed-up file; respond with *N* to leave the destination file unchanged and continue the RESTORE operation.

Warning! No files were found to restore
No files were found on the backup disk that matched the destination file specification.

RMDIR or RD

Remove Directory

2.0 and later

Internal

Purpose

Removes an empty directory from the hierarchical file structure.

Syntax

RMDIR [*drive*:][*path*]*directory_name*

or

RD [*drive*:][*path*]*directory_name*

where:

directory_name is the name of the directory to be removed, optionally preceded by a drive and/or path.

Description

The RMDIR command removes an empty directory from a disk's hierarchical file structure. The directory being deleted cannot contain any files or subdirectories (except for the special . and .. entries). The root directory or current directory of a disk cannot be deleted.

If the *path* parameter is used, it must specify a valid existing path. If no path is specified and *directory_name* is not preceded by a backslash (\), MS-DOS assumes that the directory to be removed is a subdirectory of the current directory. If no path is specified and *directory_name* is preceded by a backslash, MS-DOS assumes that the directory is a subdirectory of the root directory. The length of the full path (including the drive designator and directory name) must not exceed 63 characters.

The RMDIR command should not be used to remove subdirectories from drives affected by an ASSIGN or JOIN command. A directory affected by the SUBST command cannot be removed.

Note: If a directory contains files marked as hidden or system, that directory cannot be removed even though no files appear to exist when the directory contents are viewed using the DIR command.

Example

To remove the empty directory \LIB, which is a subdirectory of the \MSC directory on the disk in drive A, type

```
C>RMDIR A:\MSC\LIB  <Enter>
```

or

```
C>RD A:\MSC\LIB  <Enter>
```

Message

Invalid path, not directory, or directory not empty
The named directory cannot be deleted because it does not exist, some element of the path to the directory does not exist, or the directory contains files or subdirectories.

SELECT

Configure System Disk for a Specific Country

IBM

External

Purpose

Creates a system disk with time, date, and keyboard configured for a selected country. This command is available only with PC-DOS.

Syntax

SELECT [[*drive1*:] *drive2*:[*path*]] *country keyboard*

where:

drive1 is a floppy-disk drive (A or B) containing the distribution disk or, at a minimum, the PC-DOS system files, COMMAND.COM, and the FORMAT and XCOPY utilities (default = drive A) (version 3.2).

drive2 is the drive containing the disk to receive the PC-DOS system files and country information and can include a path (default = drive B) (version 3.2).

country is a code from the table below that controls the time, date, and currency formats.

keyboard is a code from the table below that controls the keyboard configuration.

Country	Country Code	Keyboard Code
Australia	061	*
Belgium	032	*
Canadian French	002	*
Denmark	045	*
Finland	358	*
France	033	FR
West Germany	049	GR
Israel	972	*
Italy	039	IT
Middle East	785	*
Netherlands	031	*
Norway	047	*
Portugal	351	*
Spain	034	SP
Sweden	046	*
Switzerland	041	*

(more)

Country	Country Code	Keyboard Code
United Kingdom	044	UK
United States	001	US

*Available only in version 3.2 and may be supplied on a separate floppy disk.

Description

The SELECT utility allows the user to create a bootable system disk configured for a particular country's keyboard layout and date, time, and currency formats without performing these steps separately.

Version 3.2 of SELECT uses the FORMAT command to format the disk in *drive2*, then uses the XCOPY command to copy all files on the disk in *drive1* (including the hidden system files) to *drive2*. If a country configuration other than one of the six KEYB*xx* utilities supplied on the distribution disk is specified, SELECT prompts the user to insert the disk containing the appropriate file.

Versions 3.0 and 3.1 of SELECT use the DISKCOPY program to copy all files on the disk in drive A (including the hidden system files) to the disk in drive B, formatting the disk if necessary.

All versions then add the appropriate CONFIG.SYS and AUTOEXEC.BAT files to the new disk to configure PC-DOS for use with the specified keyboard and country configuration. The specified configuration does not take effect until the computer is turned on or restarted using the new disk.

Examples

To create a PC-DOS system disk configured for West Germany using version 3.0 or 3.1, place a copy of the original PC-DOS distribution disk in drive A and a blank disk in drive B; then type

```
A>SELECT 049 GR  <Enter>
```

During the copy operation, the usual DISKCOPY prompts and messages are displayed. When the copy operation is complete, the two disks are compared using DISKCOMP, producing the usual DISKCOMP prompts and messages. The resulting disk includes all the files from the distribution disk (including the hidden system files), a CONFIG.SYS file that contains the line

```
COUNTRY=049
```

and an AUTOEXEC.BAT file that contains the following lines:

```
KEYBGR
ECHO OFF
CLS
DATE
TIME
VER
```

To create a PC-DOS system disk configured for West Germany using version 3.2, place a copy of the original PC-DOS distribution disk in drive A and a blank disk in drive B; then type

```
A>SELECT 049 GR  <Enter>
```

SELECT first uses the FORMAT command to format the disk in drive B, then uses XCOPY to copy all files on the distribution disk (including the system files), and finally creates a CONFIG.SYS file that contains the line

```
COUNTRY=049
```

and an AUTOEXEC.BAT file that contains the following lines:

```
PATH \;
KEYBGR
ECHO OFF
CLS
DATE
TIME
VER
```

Messages

Cannot execute *X:filename*
One of the files needed by SELECT (FORMAT, DISKCOPY, DISKCOMP, or XCOPY) is not on the source disk or is a version that is not compatible with the version of PC-DOS that is running.

File creation error
The root directory of the destination disk is full or unable to contain any more files or one of the files being created has the same name as a directory already on the destination disk.

Incorrect DOS version
The version of SELECT is not compatible with the version of PC-DOS that is running (version 3.2).

Incorrect number of parameters
Too many or too few parameters were specified in the command line or a separator character was omitted between two parameters (version 3.2).

Insert DOS diskette in drive A:
Strike any key when ready
This message prompts the user to insert the distribution disk containing the system files and COMMAND.COM into drive A (version 3.2).

Insert KEYB*xx*.COM diskette in drive *X*:
Strike any key when ready
The user responded *Y* to a previous prompt asking if KEYB*xx* is on another disk. This message prompts the user to insert that disk into the specified drive (version 3.2).

Insert target diskette in drive A:
Strike any key when ready
This message prompts the user to insert the disk that will become the country-specific system disk into drive A (versions 3.0 and 3.1).

Insert target diskette in drive B:
Strike any key when ready
This message prompts the user to insert the disk that will become the country-specific system disk into drive B (version 3.2).

Invalid country code
The country code given in the command line is not supported by this version of PC-DOS or is not a valid country code.

Invalid drive specification
One of the drives specified in the command line is invalid or does not exist in the system (version 3.2).

Invalid keyboard code
The keyboard code given in the command line is not supported by this version of PC-DOS or is not a valid keyboard code.

Invalid parameter
One of the parameters specified in the command line is invalid or is not supported by the version of SELECT that is running (version 3.2).

Invalid path
The path specified for *drive2* is invalid, contains invalid characters, or is longer than 63 characters (version 3.2).

Is KEYB*xx*.COM on another diskette (Y/N)?
The keyboard reconfiguration file for the specified country is not on the source disk. Respond with *Y* to cause SELECT to prompt for the disk containing the keyboard file after the FORMAT operation is completed; respond with *N* to terminate the SELECT command (version 3.2).

Keyboard routine not found.
The user responded *N* to a previous prompt asking if KEYB*xx* is on another disk (version 3.2).

SELECT is used to install DOS the first time. Select erases everything on the specified target and then installs DOS.
Do you want to continue (Y/N)?
This message warns the user that the specified disk will be formatted and all files on the source disk will be copied over. Respond with *Y* to continue; respond with *N* to terminate the SELECT command (version 3.2).

Unable to copy keyboard routine
An error occurred while the KEYB*xx*.COM program was being copied. Use the CHKDSK command to check the keyboard program on the source disk for damage (version 3.2).

Unable to create directory
The directory specified in the command line was not created because a directory with the same name already exists on the destination disk, the root directory of the destination disk is full, one of the directory names specified in the path does not exist, or a file with the same name already exists (version 3.2).

SET

2.0 and later

Set Environment Variable

Internal

Purpose

Defines an environment variable and a string that is its value.

Syntax

SET [*name=value*]

or

SET *name=*

where:

name	is a string of characters that defines an environment variable; lowercase letters are automatically converted to uppercase.
value	is a string of characters, a pathname, or a filename that defines the current value of *name*; no case conversion is made for *value*.

Description

The environment is a series of null-terminated ASCII (ASCIIZ) strings that contains environment variables and their values. (An environment variable associates a string consisting of a filename, a pathname, or other literal data with a symbolic name that can be referenced by programs. The form of the association is *name=value*.) The original, or master, environment belongs to the command processor and is established when the system is turned on or restarted. When a program is subsequently executed by the command processor or by another program, the new program inherits a private copy of its parent's environment.

The SET command enables the user to add, change, or delete an environment variable from the command processor's environment. If *value* is not included in the SET command, MS-DOS deletes the environment variable *name* from the environment. If the SET command is issued with no parameters, MS-DOS displays the values of all the variables in the environment.

With MS-DOS versions 2.x and 3.x, two particular variables are always found in an environment: PATH and COMSPEC. These variables are initialized during the system startup process and tell COMMAND.COM which subdirectories to search for executable files and where to find the transient portion of COMMAND.COM for reloading (versions 3.0 and later). (By default, PATH is a null string and therefore searches only the current or specified directory.) These special environment variables are influenced by the PATH and SHELL commands, respectively, but can also be changed with SET commands. Note, however, that changing the value of COMSPEC with SET will serve no useful purpose—changing to a different command processor must be done using an appropriate SHELL

command in the CONFIG.SYS file (the system must be restarted for it to take effect). Note also that it is not necessary to use the SET command with the PATH or PROMPT commands — MS-DOS will automatically add their new values to the environment if they are changed.

The environment, which can be as large as 32 KB, can be an effective source of global configuration information to executing programs. For instance, the Microsoft C Compiler and Microsoft Object Linker use environment variables to locate *include* and object library files. Environment variables can also be referenced as replaceable parameters in batch files, using the form %*name*%.

Under normal circumstances, MS-DOS expands the environment as necessary when SET commands are entered. However, when a batch file is being interpreted or when terminate-and-stay-resident (TSR) utilities have been loaded, the size of the command processor's environment becomes fixed. Under these circumstances, a SET command may result in the error message *Out of environment space.*

With version 3.2, the initial size of the environment can be increased either by using the COMMAND command with the /P and /E:*nnnn* switches at the system prompt or by including a SHELL command specifying COMMAND.COM followed by the /E:*nnnn* switch in the CONFIG.SYS file. *See* USER COMMANDS: COMMAND; CONFIG.SYS: SHELL.

Examples

To define the environment variable *USER* and set its value to *FRED*, type

```
C>SET USER=FRED  <Enter>
```

To change the value of the environment variable *USER* to *SALLY*, type

```
C>SET USER=SALLY  <Enter>
```

To delete the environment variable *USER* and its value from the environment, type

```
C>SET USER=  <Enter>
```

To display all the environment variables, type

```
C>SET  <Enter>
```

The output of this command will be in the following form:

```
COMSPEC=C:\DOS3\COMMAND.COM
PROMPT=$p$_$n$g
PATH=D:\BIN;C:\DOS3;C:\WP\WORD;C:\ASM;C:\MSC\BIN
INCLUDE=c:\msc\include;c:\windows\lib
LIB=c:\msc\lib;c:\windows\lib
TMP=c:\temp
PCF32=c:\forth\pc32
PROCOMM=c:\procomm\
```

Message

Out of environment space
The command processor's environment is full and cannot be expanded (usually because the SET command was issued from a batch file or the system has terminate-and-stay-resident [TSR] utilities installed).

SHARE

Install File-Sharing Support

3.0 and later

External

Purpose

Loads the resident file-sharing support module required by Microsoft Networks.

Syntax

SHARE [/F:*n*] [/L:*n*]

where:

/F:*n*	allocates *n* bytes of memory to hold file-sharing information (default = 2048).
/L:*n*	configures support for *n* simultaneous file-region locks (default = 20).

Description

The code that supports file sharing and locking in a networking environment is isolated in the user-installable SHARE module. After SHARE is loaded, MS-DOS checks all read and write requests against the file-sharing module. On personal computers that do not utilize network services, the SHARE module need not be loaded, leaving more memory for application programs.

The /F:*n* switch controls the amount of buffer space allocated for file-sharing information. Each open file requires the length of its full name, including the path, plus some overhead; the average pathname is approximately 20 bytes long. If the /F:*n* switch is not included in the command line, the buffer size defaults to 2048 bytes (sufficient for approximately 100 files with pathnames of average length).

The /L:*n* switch controls the number of entries to be allocated for an internal table containing file-locking information. Each active lock on a region of a file occupies one entry in the table. If the /L:*n* switch is absent, the default is support for 20 simultaneously active locks.

Example

To install the file-sharing support module, allocating 4096 bytes of space for file-sharing information and 40 file-region locks, type

```
C>SHARE /F:4096 /L:40  <Enter>
```

Messages

Incorrect DOS version
The version of SHARE is not compatible with the version of MS-DOS that is running.

Incorrect parameter
The command line included an invalid switch.

Not enough memory
System memory is insufficient to load the SHARE module or to reserve the designated file-sharing information space or file-region locks.

SHARE already installed
The SHARE command has already been executed since the system was turned on or restarted; additional executions have no effect.

SORT

Alphabetic Sort Filter

2.0 and later

External

Purpose

Reads records from standard input, sorts them alphabetically, and writes the sorted records to standard output.

Syntax

SORT [/R][/+*column*]

where:

/R	specifies a reverse, or descending, alphabetic sort.
/+*column*	specifies the first column to be used for sorting each line (default = 1).

Description

The SORT program is a filter that reads lines from standard input until an end-of-file marker is reached, sorts the lines into alphabetic order, and writes the sorted lines to standard output.

Standard input defaults to the keyboard; standard output defaults to the video display. Because standard input can be redirected, the SORT filter can also accept input from another character device, a file, or the piped output of another program or filter. (The most common use of SORT is to sort the redirected input from an ASCII text file.) Similarly, the output of SORT can be redirected to any character device or file or can be piped to another program.

SORT normally orders the lines of the input text stream alphabetically using the entire line, starting with column 1 as the sort key. Tab characters are not expanded to spaces. If the character in the sort-key column of one line is identical with the character in the sort-key column of the next line, SORT checks the next column to the right to determine which line will go before the other. If the second columns are also identical, the search continues to the right until a differing column is found. The maximum amount of data that can be sorted is 63 KB.

The /R switch causes SORT to arrange the set of lines in reverse alphabetic order. The /+*column* switch lets the user specify a column other than column 1 as the first sort key.

With versions 2.x, SORT arranges the input lines based on the ASCII value of the character in each line's sort-key column; the sort operation is therefore case sensitive. With versions 3.0 and later, SORT assigns lowercase letters the same ASCII value as uppercase letters; hence, case is effectively ignored. Depending on the COUNTRY command in effect (*see* USER COMMANDS: CONFIG.SYS: COUNTRY), versions 3.0 and later map accented characters with ASCII codes in the range 80H through 0E1H (128–225) to their unaccented equivalents for sorting.

Warning: If the output of the SORT command is redirected to a file with the same name as the input file, the contents of the input file may be destroyed.

Examples

The examples in this entry operate on an ASCII text file named RECORDS.TXT that contains the following lines:

```
Smith     Seattle
Adams     New York
Zoole     Bellevue
Jones     Boston
```

Each line of the file contains a person's surname, starting in column 1, and a city name, starting in column 10.

To sort the file RECORDS.TXT by surname and display the sorted lines on standard output, type

```
C>SORT < RECORDS.TXT  <Enter>
```

This will result in the following display:

```
Adams     New York
Jones     Boston
Smith     Seattle
Zoole     Bellevue
```

To sort the file RECORDS.TXT by surname and write the sorted lines into the file READY.DOC, type

```
C>SORT < RECORDS.TXT > READY.DOC  <Enter>
```

To sort the file RECORDS.TXT by surname in reverse alphabetic order and display the sorted lines on standard output, type

```
C>SORT /R < RECORDS.TXT  <Enter>
```

This will result in the following display:

```
Zoole     Bellevue
Smith     Seattle
Jones     Boston
Adams     New York
```

To sort the file RECORDS.TXT by city name and display the sorted lines on standard output, type

```
C>SORT /+10 < RECORDS.TXT  <Enter>
```

This will result in the following display:

```
Zoole     Bellevue
Jones     Boston
Adams     New York
Smith     Seattle
```

To use SORT as a filter to arrange a directory listing alphabetically, type

```
C>DIR ¦ SORT  <Enter>
```

To use SORT as a filter to arrange a directory listing alphabetically based on the first character of each file's extension, type

```
C>DIR ¦ SORT /+10  <Enter>
```

Messages

Invalid parameter
One of the parameters specified in the command line is invalid or the syntax is incorrect.

SORT: Incorrect DOS version
The version of SORT is not compatible with the version of MS-DOS that is running.

SORT: Insufficient disk space
The output of the SORT filter has been redirected to a file and the disk is full.

SORT: Insufficient memory
The available system memory is insufficient to run the SORT program.

SUBST

Substitute Drive for Subdirectory

3.1 and later

External No Net

Purpose

Causes a drive letter to be substituted for a directory name. SUBST is present in MS-DOS to support older application programs that do not accept pathnames.

Syntax

SUBST [*drive1*: [*drive2*:]*path*]

or

SUBST *drive1*: /D

where:

drive1	is the drive letter to be used to reference the files in *path*.
drive2	is a drive letter other than *drive1* that can optionally precede the name of the subdirectory being substituted.
path	is the subdirectory to be accessed when *drive1* is referenced, optionally preceded by *drive2*.
/D	cancels the effect of a previous SUBST command for *drive1*.

Description

The SUBST command allows a drive letter to be substituted for a subdirectory name.

The *drive1* parameter can be any valid drive letter except the current drive or *drive2*. Drive letters A through E are always available; drive letters beyond E require that an appropriate LASTDRIVE command be added to the CONFIG.SYS file and the system be restarted (*see* USER COMMANDS: CONFIG.SYS: LASTDRIVE).

After a SUBST command, the files on the disk normally referenced by *drive1* are no longer accessible. However, the files in the location specified by *path* can still be referenced by the usual methods (using their actual drive and path) as well as by the substituted drive designator.

If the SUBST command is entered without parameters, MS-DOS displays the substitutions currently in effect.

Warning: The SUBST command masks the actual disk-drive characteristics from commands that perform critical disk operations. Therefore, ASSIGN, BACKUP, CHKDSK, DISKCOMP, DISKCOPY, FDISK, FORMAT, JOIN, LABEL, and RESTORE should not be used on a drive affected by a SUBST command. CHDIR, MKDIR, RMDIR, and PATH commands that include the affected drive should be used with caution. A network drive cannot be named in a SUBST command.

Examples

To substitute drive B for the directory C:\ASM\SOURCE, type

```
C>SUBST B: C:\ASM\SOURCE  <Enter>
```

To display the substitutions currently in effect, type

```
C>SUBST  <Enter>
```

In this case, the SUBST command displays

```
B: => C:\ASM\SOURCE
```

To cancel the effect of a previous SUBST command that substituted drive B for a subdirectory, type

```
C>SUBST B: /D  <Enter>
```

Messages

Cannot SUBST a network drive
One or both of the drive parameters in the command line referred to a drive that is assigned to a network.

DOS 2.0 or later required
SUBST does not work with versions of MS-DOS earlier than 2.0.

Incorrect DOS version
The version of SUBST is not compatible with the version of MS-DOS that is running.

Incorrect number of parameters
The command line included too many or too few parameters.

Invalid parameter
The drive named in the command line is invalid, does not exist, is the default drive, or is the same as the drive in the path to be substituted.

Not enough memory
The available system memory is insufficient to run the SUBST command.

Path not found
An element of the path included in the command line is invalid or does not exist.

SYS

1.0 and later

Transfer System Files

External No Net

Purpose

Copies the hidden files that contain the operating system from the disk in the current drive to another formatted disk.

Syntax

SYS *drive*:

where:

drive is the location of the disk that will receive the system files. This parameter is required.

Description

An MS-DOS system disk must contain three files to be bootable: the two operating-system files and the command processor. The operating system itself is contained in the files IO.SYS and MSDOS.SYS (or IBMBIO.COM and IBMDOS.COM in PC-DOS), which must always be the first two files in the disk's directory. Both have file attributes set for system and hidden (all versions) and read-only (versions 2.0 and later). IO.SYS (or IBMBIO.COM) contains the default set of device drivers for the system; it must occupy contiguous sectors in the disk's files area. MSDOS.SYS (or IBMDOS.COM) contains the kernel of the operating system proper. The third required file is the shell, or command processor, which by default is COMMAND.COM. This is an unrestricted file and can be located anywhere on the disk.

The SYS command transfers the two operating-system files from the default drive to the specified destination disk. The destination disk that receives the files must meet one of the following requirements:

- The disk is formatted but completely empty.
- The disk currently contains hidden MS-DOS system files that are large enough to allow replacement by the new system files.
- The disk has been formatted with the /B switch to reserve room for the system files. (Note that /B produces a disk with only eight sectors per track.)

If the disk already contains the two hidden system files, the SYS command can be used to transfer an equivalent or later version of MS-DOS.

After the two hidden operating-system files are installed with the SYS command, the COMMAND.COM file (or another command processor) must be transferred to the destination disk with the COPY command. The resulting disk is a bootable system disk.

Note: Because the two system files have the hidden attribute, they do not appear on a directory listing produced by the DIR command. The CHKDSK command does report the presence of hidden files on a disk and will list their names if the /V switch is used but will not list such information as the file size or date and time of creation.

Example

To transfer a copy of the system files to the disk in drive B, type

```
C>SYS B:  <Enter>
```

Messages

Cannot SYS to a Network drive
The drive specified in the command line is currently assigned to a network.

Destination disk cannot be booted
The hidden operating-system files were transferred to the destination disk but could not be placed in contiguous sectors.

Incompatible system size
The destination disk already contains operating-system files and they are smaller than those being copied.

Incorrect DOS version
The version of SYS is not compatible with the version of MS-DOS that is running.

Insert destination disk in drive *X*
and strike any key when ready
This message prompts the user to insert the disk onto which the operating-system files will be copied into the specified drive.

Insert system disk in drive *X*
and strike any key when ready
This message prompts the user to insert a disk containing the operating-system files into the specified drive.

Invalid drive specification
The drive specified in the command line is invalid or does not exist in the system.

Invalid parameter
The command line contained an invalid drive letter.

No room for system on destination disk
Contiguous space at the beginning of the destination disk is insufficient for the operating-system files. This can occur when files already exist on the destination disk or when sections of the disk are marked as unusable by the FORMAT command.

No system on default drive
The disk in the default drive does not contain the two hidden system files. Replace the disk with a bootable system disk.

System transferred
The operating-system files have been successfully transferred to the destination disk.

TIME

1.0 and later

Set System Time

Internal

Purpose

Sets or displays the system time. TIME is an external command with PC-DOS version 1.0.

Syntax

TIME [*hh*:*mm*[:*ss*[.*xx*]]]

where:

hh	is hours (0-23).
mm	is minutes (0-59).
ss	is seconds (0-59).
xx	is hundredths of a second (0–99).

Note: No spaces are allowed between any of the time parameters.

Description

All computers that run MS-DOS have as part of their hardware configuration a timer, or clock, that maintains the current system date and time. One use of this clock, among others, is to insert the current date and time into a file's directory entry when the file is created or modified.

The TIME command allows the user to display or modify the current time that is being maintained by the system's real-time clock. TIME is also executed by MS-DOS when the system is turned on or restarted, unless an AUTOEXEC.BAT file is on the system disk, in which case the command is executed only if it is included in the AUTOEXEC.BAT file.

On IBM PC/ATs and compatibles, the TIME command does not permanently change the system time stored in the built-in battery-backed clock/calendar; the newly entered time is lost when the system is turned off or restarted. On these machines, the SETUP program (found on the *Diagnostics for IBM Personal Computer AT* disk or equivalent) must be used to permanently alter the clock/calendar's current time.

On IBM PCs, PC/XTs, and compatibles equipped with add-on cards containing battery-backed clock/calendar circuitry, it is usually necessary to run a time/date installation program (included with the card) to set the system date and time from the clock/calendar on the card. The TIME command generally has no effect on these card-mounted clock/calendars.

The format of times displayed by the system depends on the current country code, which is determined by the optional COUNTRY command in the CONFIG.SYS file (*see* USER COMMANDS: CONFIG.SYS: COUNTRY). The default display format is the 24-hour format (00:00–23:59).

Examples

To display the current time, type

```
C>TIME  <Enter>
```

This results in output of the following form:

```
Current time is 12:49:04.93
Enter new time:
```

To leave the time unchanged, press the Enter key.

To set the system time to 8:30 P.M., type

```
C>TIME 20:30  <Enter>
```

Messages

Current time is *hh:mm:ss.xx*
This informational message is displayed in response to any valid TIME command.

Invalid parameter
The delimiter in the time parameter included in the command line was not a colon (:) or a period (.).

Invalid time
Enter new time:
An invalid time, time format, or delimiter was specified in the command line or in response to the *Enter new time:* prompt. Note that no spaces are allowed around delimiters.

TREE

3.2

Display Directory Structure

External

Purpose

Displays the hierarchical directory structure of a disk and, optionally, the names of the files in each subdirectory. This command is included with PC-DOS beginning with version 2.0.

Syntax

TREE [*drive*:][/F]

where:

drive	is the location of the disk whose directory structure is to be displayed.
/F	displays the filenames in each directory in addition to the directory names.

Description

The TREE command displays on standard output the pathname of each directory on the disk in the specified drive, beginning with the subdirectories of the root directory. If a disk drive is not designated, TREE assumes the current, or default, drive. The name of each directory is followed by a list of its subdirectories. If the /F switch is included in the command line, the names of the files in each subdirectory are also displayed. (Prior to version 3.1, the PC-DOS TREE command does not list the files in the root directory if /F is used.)

The output of the TREE command can be redirected to another output device or a file or can be piped to another program.

Examples

Assume that the root directory of the disk in drive B contains three subdirectories: \SOURCE, \LIBS, and \DOC. The subdirectory \SOURCE in turn contains two subdirectories: \ASM and \PASCAL. To display the directory structure of this disk, type

```
C>TREE B:  <Enter>
```

The TREE command displays the following list:

```
DIRECTORY PATH LISTING FOR VOLUME MYDISK

Path: B:\SOURCE

Sub-directories:  ASM
                  PASCAL


Path: B:\SOURCE\ASM

Sub-directories:  None


Path: B:\SOURCE\PASCAL

Sub-directories:  None


Path: B:\LIBS

Sub-directories:  None


Path: B:\DOC

Sub-directories:  None
```

To display the directory structure of the disk in drive B and also display all files in each directory, type

```
C>TREE B: /F  <Enter>
```

To print the directory-structure listing of the disk in drive B on an attached printer, type

```
C>TREE B: > PRN  <Enter>
```

To display the directory structure of the disk in drive B one screenful at a time, type

```
C>TREE B: ¦ MORE  <Enter>
```

For a more compressed listing of all subdirectories on the disk in drive B, type

```
C>TREE B: ¦ FIND "Path:"  <Enter>
```

The output appears in the following form:

```
Path: B:\SOURCE
Path: B:\SOURCE\ASM
Path: B:\SOURCE\PASCAL
Path: B:\LIBS
Path: B:\DOC
```

Messages

DOS 2.0 or later required
TREE does not work with versions of MS-DOS earlier than 2.0.

Incorrect DOS version
The version of TREE is not compatible with the version of MS-DOS that is running.

Invalid drive specification
The drive specified in the command line is invalid or does not exist in the system.

Invalid parameter
The command line contained a path or filename in addition to a disk drive or contained an invalid switch.

No sub-directories exist
The specified drive has no subdirectories.

TYPE

Display File

1.0 and later

Internal

Purpose

Sends the contents of an ASCII text file to standard output.

Syntax

TYPE [*drive*:][*path*]*filename*

where:

filename is the name of the text file to be displayed, optionally preceded by a drive and/or path; wildcard characters are not permitted.

Description

The TYPE command displays the contents of a text file on standard output (usually the video display) until it encounters an end-of-file character (ASCII code 1AH). Tab characters in the file are expanded to spaces with tab stops at each eighth character position. If a file contains characters with ASCII values less than 32 or greater than 127, the resulting display includes graphics characters and other unintelligible information.

The output of the TYPE command can be redirected to another file or character device or can be piped to another program.

Examples

To display the file SHELL.C in the directory \SOURCE on the disk in drive A, type

```
C>TYPE A:\SOURCE\SHELL.C  <Enter>
```

To direct the output of the same file to the printer, type

```
C>TYPE A:\SOURCE\SHELL.C > PRN  <Enter>
```

The TYPE command can be used with the MORE filter to paginate output. For example, to display the contents of the file MENU.ASM one screenful at a time, type

```
C>TYPE MENU.ASM | MORE  <Enter>
```

Messages

File not found
The file specified in the command line cannot be found or does not exist.

Invalid drive specification
The drive specified in the command line is invalid or does not exist in the system.

Invalid path or file name
The path specified in the command line is invalid or does not exist.

VDISK.SYS

IBM

Virtual Disk

External

Purpose

Creates a virtual disk in memory. This installable driver is available only with PC-DOS.

Syntax

DEVICE=[*drive*:][*path*]VDISK.SYS [*size*] [*sector*] [*directory*] [/E] (version 3.0)

or

DEVICE=[*drive*:][*path*]VDISK.SYS [*size*] [*sector*] [*directory*] [/E[:*max*]] (version 3.1)

or

DEVICE=[*drive*:][*path*]VDISK.SYS [*comment*] [*size*] [*comment*] [*sector*] [*comment*] [*directory*] [/E[:*max*]] (version 3.2)

where:

comment	is a string of ASCII characters in the range 32 through 126, excluding the slash character (/) (version 3.2).
size	is the size of the virtual disk in kilobytes (minimum = 1, default = 64).
sector	is the sector size in bytes (128, 256, or 512; default = 128).
directory	is the maximum number of entries in the virtual disk's root directory (2–512, default = 64).
/E	causes VDISK to use extended memory.
/E:*max*	causes VDISK to use extended memory and sets the maximum number of sectors (1–8, default = 8) to transfer from extended memory at one time (versions 3.1 and later).

Note: Unless the /E switch is used, the virtual disk is created in conventional memory.

Description

The VDISK.SYS installable device driver allows the configuration of one or more virtual disks (sometimes referred to as electronic disks or RAMdisks). A virtual disk is implemented by mapping a disk's structure — directory, file allocation table, and files area — onto an area of random-access memory, rather than onto actual sectors located on a magnetic recording medium. Access to files stored in a virtual disk is very fast, because no moving parts are involved and the "disk" operates at the speed of the system's memory. (The VDISK driver is available only with PC-DOS; a similar program named RAMDRIVE.SYS is included with MS-DOS.)

Warning: Because a RAMdisk resides entirely in RAM and is therefore volatile, any information stored there is irretrievably lost when the computer loses power or is restarted.

VDISK can create a virtual disk in either conventional memory or extended memory. Conventional memory is the term for the up-to-640 KB of RAM that contain PC-DOS and any application programs. Extended memory is the term for the memory at addresses above 1 MB (100000H) that is available on 80286-based personal computers such as the IBM PC/AT.

A virtual disk can be installed in conventional memory by simply inserting the line *DEVICE=VDISK.SYS* into the system's CONFIG.SYS file and restarting the system. (If the file VDISK.SYS is not in the root directory of the startup disk, it may be preceded by a drive and/or path.) A new "drive" then becomes available in the system, with default values of 64 KB disk size, 128-byte sectors, and 64 available directory entries (assuming there is sufficient memory). The virtual disk is assigned the next available drive letter (which is displayed in VDISK's sign-on message). The drive letter assigned depends on the number of other physical and virtual disks in the system and also on the position of the *DEVICE=VDISK.SYS* line in the CONFIG.SYS file relative to other installed block devices. Available memory permitting, multiple virtual disks can be created by using multiple *DEVICE=VDISK.SYS* lines. Several optional parameters allow the user to customize the size and configuration of the virtual disk and to use extended memory if it is available.

The *size* parameter specifies the amount of RAM, in kilobytes, to be allocated to the virtual disk. The default is 64 KB, but any size from 1 KB to the total amount of available memory can be specified. If the size specified is greater than available memory or less than 1 KB, VDISK ignores it and creates a virtual disk of 64 KB. If necessary, VDISK also adjusts the *size* value to ensure that at least 64 KB of memory remain available in the system.

The *sector* parameter sets the virtual sector size used within the virtual disk. The *sector* value may be 128, 256, or 512 bytes (default = 128 bytes). Selection of the smallest sector size results in a minimum of wasted virtual disk space per file but also results in somewhat slower transfer of data.

Note: Physical disk devices in IBM PC-compatible systems always use 512-byte sectors.

The *directory* parameter sets the number of available entries in the virtual disk's root directory. The allowed range is 2 through 512 (default = 64). Each directory entry requires 32 bytes. VDISK rounds the number of available directory entries up, if necessary, so that an integral number of sectors are assigned to the root directory.

The /E switch causes VDISK to use extended memory for the virtual disk, rather than conventional memory. This allows very large virtual disks to be configured while still leaving the maximum amount of conventional memory available for use by application programs. If the /E switch is used and extended memory is not present in the system, the VDISK driver will not install itself.

When /E is used in the form /E:*max*, the variable *max* controls how many virtual sectors can be transferred at a time from extended memory. The value of *max* must be in the range 1 through 8 (default = 8). If VDISK operation appears to conflict with the communications port or other interrupt-driven peripheral devices, the *max* variable should be set to a smaller number. The *max* option is available only with versions 3.1 and 3.2.

Note: If VDISK uses conventional memory for virtual disk storage, the memory cannot be reclaimed except by modifying the CONFIG.SYS file and restarting the system.

Examples

To create a virtual disk drive with the default values of 64 KB disk size, 128-byte sectors, and 64 available directory entries, include the command

```
DEVICE=VDISK.SYS
```

in the CONFIG.SYS file and restart the system.

To create a 360 KB virtual disk with 512-byte sectors and 112 available directory entries when the file VDISK.SYS is located in a directory named \BIN on drive C, include the command

```
DEVICE=C:\BIN\VDISK.SYS 360 512 112
```

in the CONFIG.SYS file and restart the system. The directory for this virtual disk requires 3584 bytes (112 entries * 32 bytes), or 7 sectors.

With version 3.2, comments can be inserted between the values to identify them. For example, to create a 1 MB virtual disk drive in extended memory with 256-byte sectors and 128 directory entries, placing comments before the values to identify them, include the command

```
DEVICE=VDISK.SYS DISK_SIZE: 1024 SECTOR_SIZE: 256 DIR_ENTRIES: 128 /E
```

in the CONFIG.SYS file and restart the system.

Messages

Buffer size adjusted
No *size* value was specified or the specified value was larger than the amount of available memory.

Directory entries adjusted
No *directory* value was specified, VDISK adjusted the *directory* value up to the nearest sector-size boundary, or the *size* value was too small to hold the file allocation table, the directory, and two additional sectors, in which case VDISK adjusted *directory* downward until these conditions were met.

Invalid switch character
A slash character (/) was included in a comment or the /E switch was entered incorrectly.

Sector size adjusted
The *sector* value was missing from the command line or an incorrect value was entered; therefore, VDISK used the default value of 128 bytes.

Transfer size adjusted
A value outside the range 1 through 8 was specified with the /E:*max* switch; therefore, VDISK used the default value of 8.

VDISK not installed - Extender Card switches do not match the system memory size
The switch settings on the extender card are not correct or the extended memory exists in an expansion unit, which VDISK is not capable of using.

VDISK not installed - insufficient memory
Less than 64 KB of system memory remained after attempted installation, the /E switch was specified and the system does not contain extended memory, or the amount of available extended memory was too small to support the installation of VDISK.

VDISK Version *n.nn* virtual disk *X*:
Buffer size: *nn* KB
Sector size: *nnn*
Directory size: *nnn*
Transfer size: *n*
VDISK was successfully installed and this message informs the user of the drive letter assigned to the virtual disk, the version of VDISK that created the disk, and the characteristics of the disk. The *Transfer size:* message appears only in versions 3.1 and 3.2 and only if the /E switch was used.

VER

2.0 and later

Display Version

Internal

Purpose

Displays the MS-DOS version number.

Syntax

VER

Description

The VER command displays on standard output (usually the video display) the number of the MS-DOS version that is running. The version number is also displayed as part of the copyright notice when the system is turned on or restarted, unless an AUTOEXEC.BAT file is on the system disk. (The VER command can be included in the AUTOEXEC.BAT file to display the version number, but it will not display the copyright information.)

Examples

To display the MS-DOS version number, type

```
C>VER  <Enter>
```

On a system that is running MS-DOS version 3.2, the following message is displayed:

```
MS-DOS Version 3.2
```

To print the MS-DOS version number on an attached printer instead of displaying it on the screen, type

```
C>VER > PRN  <Enter>
```

VERIFY

Set Verify Flag

2.0 and later

Internal

Purpose

Sets the system's internal flag controlling verification of disk writes.

Syntax

VERIFY [ON|OFF]

Description

The VERIFY command sets or clears an internal MS-DOS flag that controls verification of data written to disks. (The actual verification process is usually carried out by the device driver and the disk-drive controller.) The VERIFY ON command has the same effect on a global basis as the /V switch has on COPY operations. (When VERIFY is on, use of the /V switch with COPY has no additional effect.) VERIFY ON remains in effect until a program turns it off with a Set Verify system call or until the user types *VERIFY OFF* at the command prompt. The VERIFY command does not affect the operation of character devices.

When the VERIFY command is entered without an ON or OFF, MS-DOS displays the current state of the system's internal verify flag. The default setting of the verify flag is off.

Examples

To turn on verification of disk writes, type

```
C>VERIFY ON  <Enter>
```

To display the current status of the verify flag, type

```
C>VERIFY  <Enter>
```

Messages

Must specify ON or OFF
The command line contained an invalid parameter.

VERIFY is off

or

VERIFY is on
No setting was specified in the command line and VERIFY displays this informational message indicating the current status of the verify flag.

VOL

2.0 and later

Display Disk Name

Internal

Purpose

Displays a disk's volume label if one exists.

Syntax

VOL [*drive*:]

where:

drive is the location of the disk whose volume label is to be displayed.

Description

The VOL command displays a disk's name, or volume label. If *drive* is not included in the command line, the volume label of the disk in the current drive is displayed.

A volume label can be assigned to a disk when it is formatted by using the /V switch with the FORMAT command. A volume label can be added, changed, or deleted *after* a disk has already been formatted by using the LABEL command (PC-DOS versions 3.0 and later, MS-DOS versions 3.1 and later). The CHKDSK, DIR, and TREE commands also display a disk's volume label as part of their output.

Example

To display the volume label for the disk in the current drive, type

```
C>VOL  <Enter>
```

If the disk's name is HARDDISK, the VOL command produces the following output:

```
Volume in drive C is HARDDISK
```

Messages

Invalid drive specification
The drive specified in the command line is invalid or does not exist in the system.

Volume in drive *X* has no label
The disk in the current or specified drive was not previously assigned a volume label with the FORMAT or LABEL command.

XCOPY

Copy Files

3.2

External

Purpose

Copies files and directories, optionally also copying subdirectories and the files they contain.

Syntax

XCOPY *source* [*destination*][/A] [/D:*mm-dd-yy*] [/E] [/M] [/P] [/S] [/V] [/W]

where:

source	is the name of the file(s) to be copied, optionally preceded by a drive and/or path; wildcard characters are permitted in the filename. If the path is omitted, a drive letter must be specified; this parameter is not optional.
destination	is the destination location and, optionally, the name for the copied files, and can be preceded by a drive; wildcard characters are permitted in the filename.
/A	copies only those source files with the archive bit set.
/D:*mm-dd-yy*	copies only files modified on or after the specified date. (The date format depends on the COUNTRY command in effect, if any.)
/E	copies empty subdirectories; if this switch is used, the /S switch must also be specified.
/M	copies only those files with the archive bit set; also turns off the archive bit of each source file after it is copied.
/P	prompts the user for confirmation before copying each file.
/S	copies all nonempty subdirectories of *source* and the files they contain.
/V	performs read-after-write verification of destination file(s).
/W	waits for the user to press a key before copying any files, allowing disks to be changed.

Description

The XCOPY command copies one or more source files to one or more destination files. Unlike the COPY command, however, a single XCOPY command can copy *all* files contained in the entire hierarchical file structure of the source disk to the destination disk, creating a corresponding set of directories and subdirectories at the destination to hold the copied files.

The *source* parameter identifies the file or files to be copied. It can consist of any combination of a drive, path, and filename (optionally including wildcards) but *must* include either

a drive or a pathname. If only a drive is specified, all files in the current directory of that drive are copied. If a path without a drive or filename is specified, all files in the named directory are copied from the current drive.

The *destination* parameter can also consist of any combination of drive, path, and filename. Unless only a single file is being copied and it is also being renamed as part of the XCOPY operation, *destination* is usually simply a drive and/or path specifying where to place the copied file. If *destination* includes a filename, XCOPY displays a message asking if the specified destination is a file or a directory. Depending on the user's response, XCOPY then either copies the source file to a destination file with the specified name or creates a directory with the specified name and copies the source files into it. (Note that if the user responds that the destination is to be a file and multiple source files were specified in the command line, only the last source file is copied to the specified destination.) If no destination is specified, the source file is copied to a file with the same name in the current directory of the current drive.

The /A, /D:*mm-dd-yy*, /M, and /P switches allow selective copying of files. The /A switch is used to copy only source files with the archive bit set; the /M switch also copies only source files with the archive bit set but turns off each source file's archive bit after the file is copied. The /D:*mm-dd-yy* switch is used to copy files that were modified on or after a selected date; the date must be entered in one of the formats discussed in the entry for the system's DATE command or in the format of the COUNTRY command currently in effect (*see* USER COMMANDS: CONFIG.SYS: COUNTRY). The /P switch causes XCOPY to prompt the user for confirmation before transferring each file.

The /E and /S switches allow an entire branch of the source disk's hierarchical directory structure to be copied. If the /S switch is specified, XCOPY copies all nonempty subdirectories of *source*, creating equivalent destination subdirectories, if necessary, to hold the files. If the /E switch is specified, XCOPY also duplicates empty source subdirectories in the equivalent destination locations. If the /E switch is used, the /S switch must also be specified.

The /V switch causes a Verify call to be issued on the destination file(s) to ensure that the data was written correctly. Its effect is equivalent to that of the VERIFY ON command.

Finally, the /W switch causes XCOPY to wait for the user to press a key before copying any files, thus allowing an exchange of disks before the files are transferred. This is useful in systems without a fixed disk, because it allows XCOPY to be used when the program itself is not on either the source or the destination disk.

Note: With MS-DOS versions of XCOPY, the related program MCOPY can be created by simply copying the file XCOPY.EXE to a file named MCOPY.EXE using the following command:

```
C>COPY /B XCOPY.EXE MCOPY.EXE  <Enter>
```

What distinguishes MCOPY from XCOPY is the program name; when either program is loaded, it looks at the name under which it was invoked and reconfigures itself accordingly. MCOPY's behavior is similar to XCOPY's, except that MCOPY automatically

determines whether the name specified as the destination is a file or a directory according to the following rules:

- If the source is a directory, the specified destination is a directory.
- If the source includes multiple files, the specified destination is a directory.
- If the destination name ends with a backslash character (\), the specified destination is a directory.

MCOPY supports all the XCOPY switches.

Not all implementations of XCOPY can be renamed to MCOPY and function accordingly. The PC-DOS version of XCOPY, for example, does not support this feature.

Return Codes

0 No errors were detected during the copy operation.
1 No files were found to copy.
2 The copy operation was terminated by a Ctrl-C or Ctrl-Break.
4 Initialization error occurred: not enough memory, file not found, or command-line syntax error.
5 The copy operation was terminated by an *A* response to an *Abort, Retry, Ignore?* prompt.

Examples

To copy all files in the directory C:\SOURCE to the directory C:\SOURCE\BACKUP, type

```
C>XCOPY C:\SOURCE\*.* C:\SOURCE\BACKUP  <Enter>
```

To copy all files and directories on drive C to the disk in drive D, type

```
C>XCOPY C:\*.* D: /S /E  <Enter>
```

Messages

***nn* File(s) copied**
This informational message is displayed at the completion of an XCOPY command and indicates the total number of source files processed.

***filename* File not found**
The source file specified in the command line is invalid or does not exist.

***X:pathname* (Y/N)?**
The /P switch was specified in the command line. XCOPY displays the name of each file, preceded by a drive (and path, if one was specified), and asks for confirmation before copying the file.

Access denied
A destination file could not be overwritten because it was marked read-only.

Cannot COPY from a reserved device
A character device such as AUX or COM1 cannot be the source of an XCOPY operation.

Cannot COPY to a reserved device
A character device such as PRN cannot be the destination of an XCOPY operation.

Cannot perform a cyclic copy
The command line included a /S switch and the destination directory is a subdirectory of the source directory. A subdirectory cannot be copied onto itself.

Does *name* specify a file name or directory name on the target (F = file, D = directory)?
The specified destination directory does not already exist; the user is prompted to determine whether it should be created. Respond with *F* to copy the source file to a file named *name*; respond with *D* to create a subdirectory named *name* and copy the source file into it.

File cannot be copied onto itself
The name and location of the source file are the same as the name and location of the destination file.

File creation error
A destination file or directory could not be created. The destination disk may be full.

Incorrect DOS version
The version of XCOPY is not compatible with the version of MS-DOS that is running.

Insufficient disk space
The disk does not contain enough available space to perform the specified XCOPY operation.

Insufficient memory
The available system memory is insufficient to perform the XCOPY operation.

Invalid date
The command included a /D switch and the date was not formatted properly.

Invalid drive specification
The source or destination drive specified in the command line is not valid or does not exist in the system.

Invalid number of parameters
The command line contained too many or too few filenames or other parameters.

Invalid parameter
A switch supplied in the command line is not valid.

Invalid path
A directory specified in the command line is invalid or does not exist.

Lock Violation
XCOPY attempted to access a file in use by another program. Respond with *A* to the error-message prompt and try XCOPY later or wait for a few minutes and respond with *R*.

Path not found
One of the pathnames specified in the command line is invalid or does not exist.

Path too long
The path element of the source or destination parameter was longer than 63 characters.

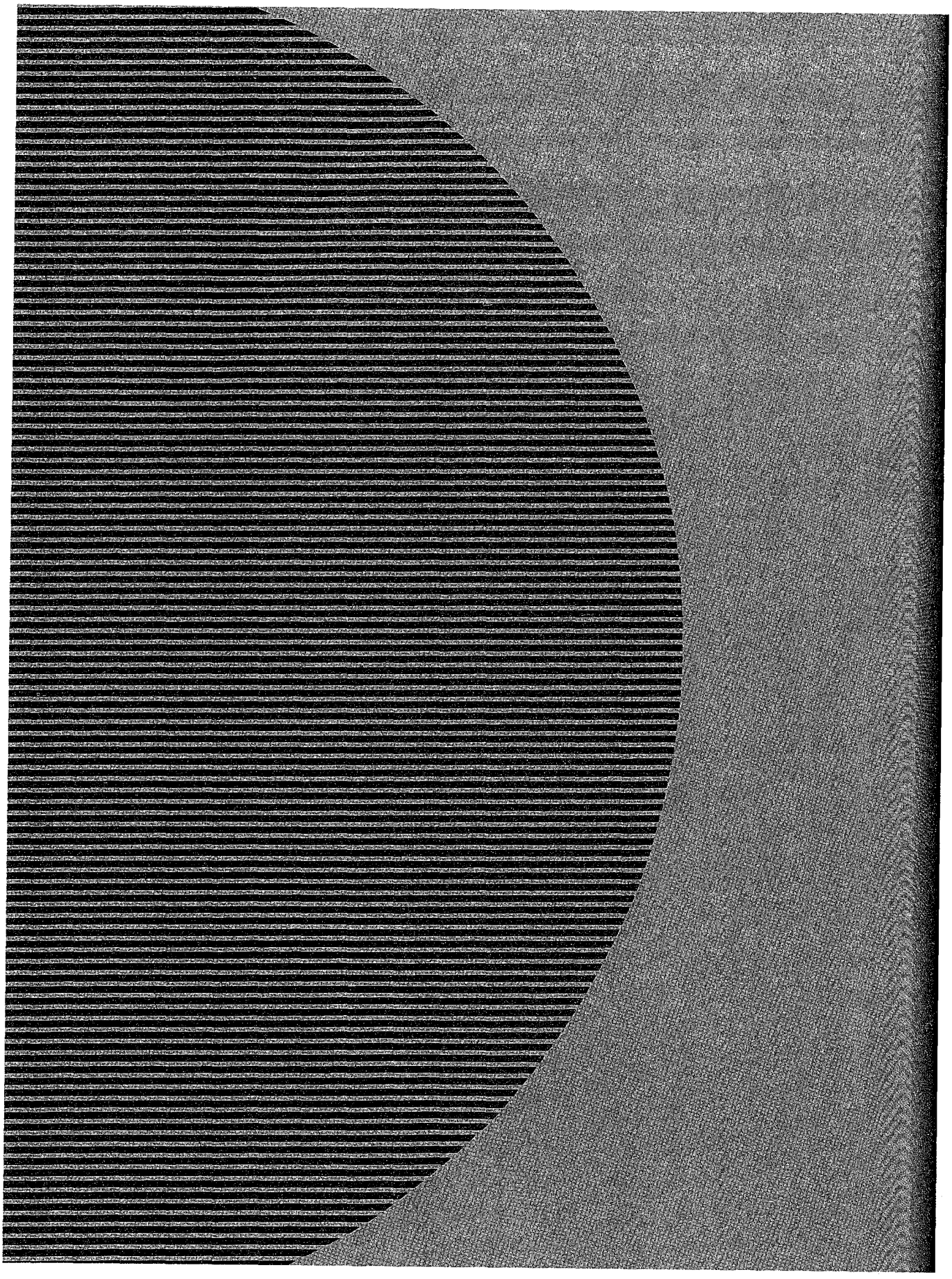
Press any key to begin copying file(s)
The /W switch was specified in the command line and XCOPY waits for the user to press a key before beginning the copy process.

Reading source file(s)...
This informational message is displayed during the XCOPY operation.

Sharing violation
XCOPY attempted to access a file in use by another program. Respond with *A* to the error-message prompt and try XCOPY later or wait a few minutes and respond with *R*.

Too many open files
XCOPY failed due to a lack of available system file handles. Increase the size of the FILES command in the CONFIG.SYS file, restart the system, and attempt the XCOPY command again.

Unable to create directory
A destination directory cannot have the same name as an existing file in the prospective parent directory.



Section IV
Programming Utilities

Introduction

This section of *The MS-DOS Encyclopedia* describes the Microsoft utilities, documentation aids, and debuggers that can be used with the Microsoft C, FORTRAN, Pascal, and BASIC compilers and with the Microsoft Macro Assembler (MASM). Included are operating instructions for MASM, the Macro Assembler; LIB, the Library Manager; LINK, the Microsoft Object Linker; the DEBUG, SYMDEB, and CodeView program debuggers; MAKE, which automates maintenance of programs; CREF, which produces a cross-reference listing of symbols; and EXE2BIN, EXEMOD, and EXEPACK, which modify executable files.

Entries (except for the program debuggers) are arranged alphabetically by the name of the programming utility. The three Microsoft debuggers are listed at the end of the section in the following order: DEBUG, SYMDEB, CodeView. Individual DEBUG and SYMDEB commands appear alphabetically under the headings DEBUG and SYMDEB.

Each utility entry includes

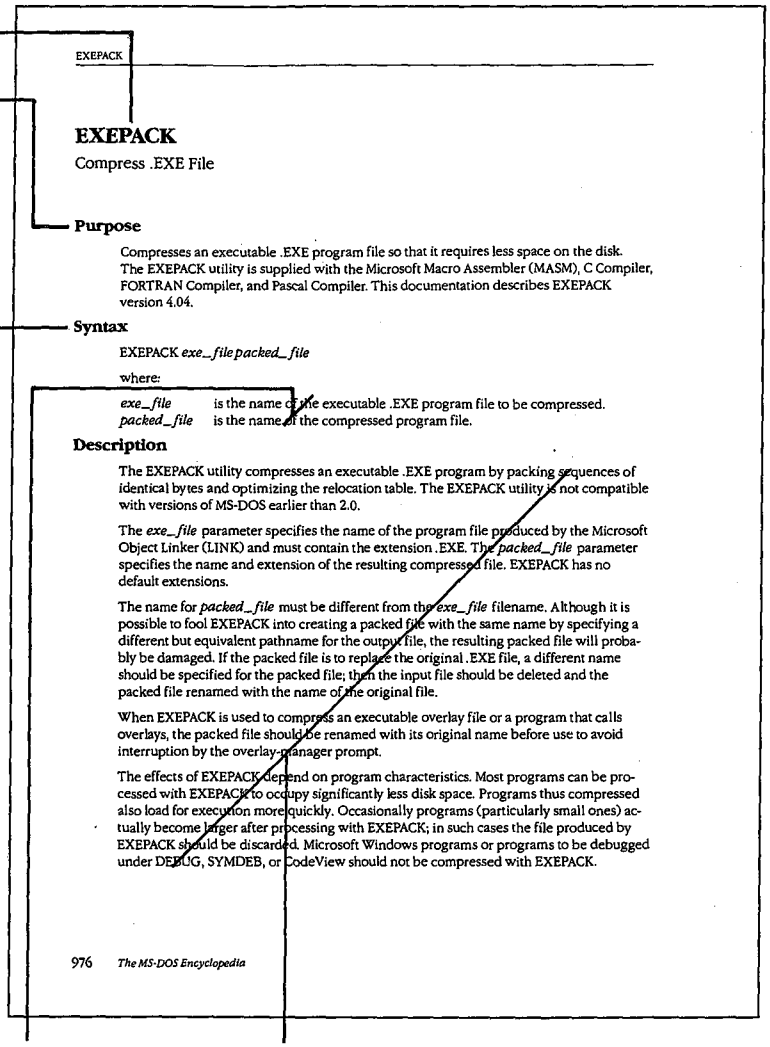
- Utility name
- Utility purpose
- Prototype command line and summary of options
- Detailed description of utility
- One or more examples of utility use
- Return codes (where applicable)
- Error messages and warnings (where applicable)

The experienced user can find information with a quick glance at the first part of a utility entry; a less experienced user can refer to the detailed explanation and examples in a more leisurely fashion. The next two pages contain an example of a typical entry from the Programming Utilities section, with explanations of each component.

HEADING
The utility name.

PURPOSE
An abstract of utility purpose and usage plus a statement of which Microsoft products the utility is supplied with and the utility version described in the entry.

SYNTAX
A prototype command line, with variable names in italic and optional parameters in square brackets. The various elements of the command line should be entered in the order shown. Any punctuation must be used exactly as shown; in commands that use commas as separators, the comma usually must be included as a placeholder even if the parameter is omitted. Except where noted, commands, parameters, and switches can be entered in either uppercase or lowercase. Utility names can be preceded by a drive and/or path.



EXEPACK

EXEPACK

Compress .EXE File

Purpose

Compresses an executable .EXE program file so that it requires less space on the disk. The EXEPACK utility is supplied with the Microsoft Macro Assembler (MASM), C Compiler, FORTRAN Compiler, and Pascal Compiler. This documentation describes EXEPACK version 4.04.

Syntax

EXEPACK *exe_file packed_file*

where:

exe_file is the name of the executable .EXE program file to be compressed.
packed_file is the name of the compressed program file.

Description

The EXEPACK utility compresses an executable .EXE program by packing sequences of identical bytes and optimizing the relocation table. The EXEPACK utility is not compatible with versions of MS-DOS earlier than 2.0.

The *exe_file* parameter specifies the name of the program file produced by the Microsoft Object Linker (LINK) and must contain the extension .EXE. The *packed_file* parameter specifies the name and extension of the resulting compressed file. EXEPACK has no default extensions.

The name for *packed_file* must be different from the *exe_file* filename. Although it is possible to fool EXEPACK into creating a packed file with the same name by specifying a different but equivalent pathname for the output file, the resulting packed file will probably be damaged. If the packed file is to replace the original .EXE file, a different name should be specified for the packed file; then the input file should be deleted and the packed file renamed with the name of the original file.

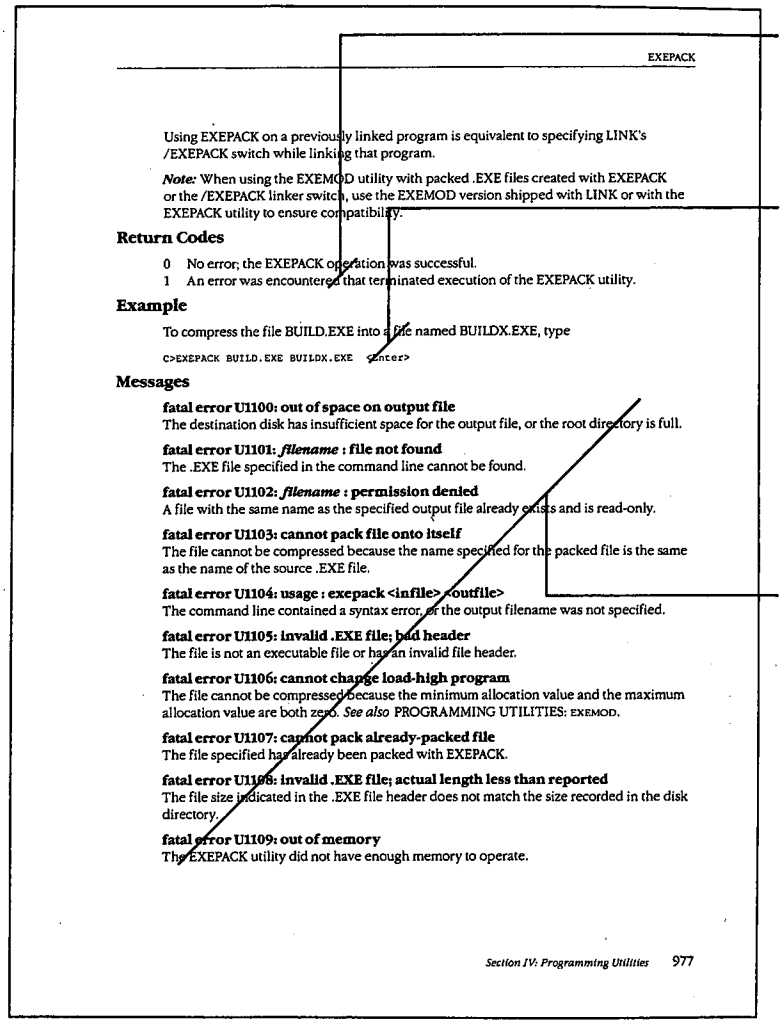
When EXEPACK is used to compress an executable overlay file or a program that calls overlays, the packed file should be renamed with its original name before use to avoid interruption by the overlay-manager prompt.

The effects of EXEPACK depend on program characteristics. Most programs can be processed with EXEPACK to occupy significantly less disk space. Programs thus compressed also load for execution more quickly. Occasionally programs (particularly small ones) actually become larger after processing with EXEPACK; in such cases the file produced by EXEPACK should be discarded. Microsoft Windows programs or programs to be debugged under DEBUG, SYMDEB, or CodeView should not be compressed with EXEPACK.

976 *The MS-DOS Encyclopedia*

BELOW WHERE
A brief explanation of each command parameter and switch. Filenames are always listed first, followed by the switches in alphabetic order. Any special position required for a filename or switch is shown in the syntax line and noted in the explanation.

DESCRIPTION
A detailed description of the utility, including a full explanation of default values, possible interactions of command parameters and options, useful background information, and any applicable warnings.



EXEPACK

Using EXEPACK on a previously linked program is equivalent to specifying LINK's /EXEPACK switch while linking that program.

Note: When using the EXEMOD utility with packed .EXE files created with EXEPACK or the /EXEPACK linker switch, use the EXEMOD version shipped with LINK or with the EXEPACK utility to ensure compatibility.

Return Codes

0 No error; the EXEPACK operation was successful.
1 An error was encountered that terminated execution of the EXEPACK utility.

Example

To compress the file BUILD.EXE into a file named BUILDX.EXE, type

```
C>EXEPACK BUILD.EXE BUILDX.EXE <Enter>
```

Messages

fatal error U1100: out of space on output file
The destination disk has insufficient space for the output file, or the root directory is full.

fatal error U1101: *filename* : file not found
The .EXE file specified in the command line cannot be found.

fatal error U1102: *filename* : permission denied
A file with the same name as the specified output file already exists and is read-only.

fatal error U1103: cannot pack file onto itself
The file cannot be compressed because the name specified for the packed file is the same as the name of the source .EXE file.

fatal error U1104: usage : exepack <infile> <outfile>
The command line contained a syntax error, or the output filename was not specified.

fatal error U1105: invalid .EXE file; bad header
The file is not an executable file or has an invalid file header.

fatal error U1106: cannot change load-high program
The file cannot be compressed because the minimum allocation value and the maximum allocation value are both zero. *See also* PROGRAMMING UTILITIES: EXEMOD.

fatal error U1107: cannot pack already-packed file
The file specified has already been packed with EXEPACK.

fatal error U1108: invalid .EXE file; actual length less than reported
The file size indicated in the .EXE file header does not match the size recorded in the disk directory.

fatal error U1109: out of memory
The EXEPACK utility did not have enough memory to operate.

Section IV: Programming Utilities 977

RETURN CODES
Exit codes returned by the utility (if any) that can be tested in a batch file or by another program.

EXAMPLES
One or more examples of the utility at work, including examples of the resulting output where appropriate. User entry appears in color; do not type the prompt, which appears in black. Press the Enter key (labeled Return on some keyboards) as directed at the end of each command line.

MESSAGES
An alphabetic list of messages that may be displayed when the utility is used. Following each message is a brief explanation of the condition that produces the message and, where appropriate, any action that should be taken.

CREF

Generate Cross-Reference Listing

Purpose

Produces a cross-reference listing of all symbols in an assembly-language program. The CREF utility is supplied with the Microsoft Macro Assembler (MASM). This documentation describes CREF version 4.0.

Syntax

CREF

or

CREF *crf_file*[;]

or

CREF *crf_file,ref_file*

where:

crf_file is the input file previously produced by MASM (default extension = .CRF).
ref_file is the output ASCII text file to be created (default extension = .REF).

Description

The CREF utility processes a file produced by MASM and generates an ASCII cross-reference listing in a file on disk or directly on a character device (such as a printer). The output file contains an alphabetic list of the symbols in the assembled program, including the line number of each reference to the symbol and the total number of symbols in the program. A pound sign (#) follows the line number of the reference that defines the symbol.

The *crf_file* has the default extension .CRF. It is produced by providing MASM with a file-name other than NUL in the cross-reference position in the command line, by responding to the *Cross-reference:* prompt, or by including the /C switch in the MASM command line or at any MASM prompt. An assembly source listing file (.LST) must also be requested in the MASM command line or in response to the MASM prompts in order to generate a valid .CRF file.

If a semicolon follows the *crf_file* parameter in the CREF command, the resulting *ref_file* containing the cross-reference listing is given the same drive and pathname as *crf_file*, with a .REF extension. If the optional *ref_file* parameter is present, it can consist of any pathname with an optional extension (default is .REF). The cross-reference listing can be sent directly to a character device, rather than to a file, by specifying a valid character device name (such as PRN) in the *ref_file* position.

If the CREF utility is run without any parameters or with some parameters missing, the CREF utility prompts the operator for the necessary information.

Return Codes

0 No error; the CREF operation was successful.
1 An error was encountered that terminated execution of the CREF utility.

Examples

To process the file MENUMGR.CRF (created during assembly of MENUMGR.ASM) into the cross-reference file MENUMGR.REF, type

```
C>CREF MENUMGR;  <Enter>
```

To process the file MENUMGR.CRF and assign the name MENU.REF to the resulting cross-reference file, type

```
C>CREF MENUMGR,MENU  <Enter>
```

To process the file MENUMGR.CRF and send the cross-reference listing directly to the printer, type

```
C>CREF MENUMGR,PRN  <Enter>
```

To run the CREF program in interactive mode, type

```
C>CREF  <Enter>
```

The following is an example of an interactive CREF session:

```
C>CREF  <Enter>
Microsoft (R) Cross Reference Utility  Version 4.00
Copyright (C) Microsoft Corp 1981, 1983, 1984, 1985.  All rights reserved.

Cross-reference [.CRF]: MENUMGR  <Enter>
Listing [MENUMGR.REF]:  <Enter>

9 Symbols

C>
```

The following sequence of commands produces the cross-reference listing HELLO.REF from the assembly-language source file HELLO.ASM:

```
C>MASM HELLO,HELLO,HELLO,HELLO  <Enter>
C>CREF HELLO;  <Enter>
```

Contents of the file HELLO.ASM:

```
        name    hello
        page    55,132
        title   HELLO.ASM - print Hello on terminal
;
; HELLO.COM utility to demonstrate CREF listing
;
cr      equ     0dh             ;ASCII carriage return
lf      equ     0ah             ;ASCII linefeed

cseg    segment para public "CODE"

        org     100h

        assume  cs:cseg,ds:cseg,es:cseg,ss:cseg

print   proc    near
        mov     dx,offset message
        mov     ah,9            ;print the string "Hello"
        int     21h
        mov     ax,4c00h        ;exit to MS-DOS
        int     21h             ;with "return code" of zero
print   endp

message db      cr,lf,'Hello!',cr,lf,'$'

cseg    ends

        end     print
```

Contents of the file HELLO.REF:

```
Microsoft Cross-Reference  Version 4.00              Mon Sep 07 23:31:21 1987
HELLO.ASM - print Hello on terminal

  Symbol Cross-Reference           (# is definition)                    Cref-1

CODE . . . . . .  10
CR . . . . . .  7      7#    24     25
CSEG . . . . . .  10     10#    14     14     14     14     27

LF . . . . . .  8      8#    24     25

MESSAGE. . . . . .  17     24     24#

PRINT. . . . . .  16     16#    29


 6 Symbols
```

Messages

can't open cross-reference file for reading
The pathname or drive specified for the input .CRF file is invalid or does not exist.

can't open listing file for writing
A write error has halted the creation of the .REF listing file. This indicates that the disk is full or write-protected, that the specified output file is read-only, or that the specified device is not available.

cref has no switches
A switch was specified in the command line; CREF has no optional switches.

DOS 2.0 or later required
CREF does not work with versions of MS-DOS earlier than 2.0.

extra file name ignored
More than two filenames were specified in the command line. The CREF utility generates the cross-reference listing using the first two filenames specified.

line invalid, start again
No .CRF file was specified in the command line or at the prompt. Specify a valid .CRF file at the prompt following this message.

out of heap space
Memory is insufficient to process the .CRF file. Remove memory-resident programs and shells or add more memory.

premature eof
The input file specified is damaged or is not a valid .CRF file.

read error on stdin
A Control-Z was received from the keyboard or a redirected file and has halted CREF.

EXE2BIN

Convert .EXE File to Binary-Image File

Purpose

Converts an executable file in the .EXE format to a memory-image file in binary format. The EXE2BIN utility is supplied with the MS-DOS distribution disks.

Syntax

EXE2BIN *exe_file* [*bin_file*]

where:

exe_file	is the .EXE-format file to be converted (default extension = .EXE).
bin_file	is the name to be given to the converted file (default extension = .BIN).

Description

The .EXE executable program files produced by the Microsoft Object Linker (LINK) contain a special header and a relocation table as well as the program code and data. The EXE2BIN utility can be used to convert a .EXE file to a .COM executable file, which is an absolute memory image of the program to be executed and does not contain a special header or relocation table. The EXE2BIN utility can also be used to convert .EXE files with an origin of zero (such as installable MS-DOS device drivers) to pure memory-image files. Files in memory-image format (a common format for device drivers and for programs to be placed in ROM for execution) usually have a .BIN or .SYS extension.

To convert a .EXE program to a binary-image file, the following are required:

- The program must be a valid .EXE file produced by LINK.
- The program can contain only one segment and cannot contain a declared stack segment.
- The program code and data portion of the .EXE file must be less than 64 KB.

To convert a .EXE program to an executable .COM file, the following are required:

- The origin of the program must be 0100H, which must also be specified as the entry point.
- The program code and data portion of the .EXE file must be less than 65227 bytes (64 KB minus 256 bytes used by the program segment prefix minus 2 bytes initially placed on the stack).
- The program must not include any FAR references.

Note: Many compilers cannot create programs that can be converted to .COM files. Check the compiler documentation for specific information concerning executable .COM files.

The *exe_file* parameter in the command line can have any filename and can include a drive and path; the default extension is .EXE. The optional *bin_file* parameter can also contain any filename and a drive and path; the default extension is .BIN. If no path is specified with the *bin_file* parameter, the output file is given the same drive and path as the *exe_file*. If no *bin_file* parameter is supplied, the output file is given the same name as the *exe_file*, with the extension .BIN.

If the program in the .EXE file requires segment fixups (that is, if the program contains instructions requiring segment relocation, which would ordinarily be done by the MS-DOS loader using the .EXE file's relocation table), EXE2BIN prompts for a base segment address. When segment fixups are necessary, the resulting program is not relocatable and must be loaded at the given location to be executed; the MS-DOS loader cannot load the program.

Examples

To convert the file HELLO.EXE to the file HELLO.BIN, type

```
C>EXE2BIN HELLO  <Enter>
```

To convert the file CLEAN.EXE, which has an origin of 0100H and meets the requirements for an executable .COM file, to the file CLEAN.COM, type

```
C>EXE2BIN CLEAN.EXE CLEAN.COM  <Enter>
```

To convert the file ASYNCH.EXE, produced by assembling and linking the device-driver source file ASYNCH.ASM, to the installable device-driver file ASYNCH.SYS, type

```
C>EXE2BIN ASYNCH.EXE ASYNCH.SYS  <Enter>
```

Messages

File cannot be converted
The program to be converted has one of the following problems: The program has an origin of 0100H but a different entry point; the program requires segment fixups; the program code and data are larger than 64 KB; the program has more than one declared segment; or the file is not a valid .EXE-format file.

File creation error
EXE2BIN cannot create the output file because a read-only file with the same name already exists, because the specified directory is full, or because the specified disk is full, write-protected, or unreadable.

File not found
The file does not exist or the incorrect path was given.

Fixups needed - base segment (hex):
The .EXE-format file contains segment references that would ordinarily be relocated by the .EXE file loader. Specify the absolute segment address at which the converted module will be executed.

Incorrect DOS version
The version of EXE2BIN is not compatible with the version of MS-DOS that is running.

Insufficient disk space
The destination disk has insufficent space to create the memory-image output file.

Insufficient memory
Not enough memory is available to run EXE2BIN.

WARNING - Read error in EXE file.
Amount read less than size in header.
The file size given in the .EXE header is inconsistent with the actual size of the file.

EXEMOD

Modify .EXE File Header

Purpose

Allows inspection or modification of the fields in a .EXE file header. The EXEMOD utility is supplied with the Microsoft Macro Assembler (MASM), C Compiler, FORTRAN Compiler, and Pascal Compiler. This documentation describes EXEMOD version 4.02.

Syntax

EXEMOD *exe_file*[/H]

or

EXEMOD *exe_file*[/STACK *n*][/MAX *n*][/MIN *n*]

where:

exe_file	is the name of an executable program in .EXE format (the extension .EXE is assumed).
/H	displays the values in the file's header.
/STACK *n*	modifies the size of the program's stack segment to *n* (hexadecimal) bytes.
/MAX *n*	sets the maximum memory allocation for the program to *n* (hexadecimal) paragraphs.
/MIN *n*	sets the minimum memory allocation for the program to *n* (hexadecimal) paragraphs.

Note: Switches can be either uppercase or lowercase and can be preceded by a dash (-) instead of a forward slash (/).

Description

Programs that are executable under MS-DOS can be in one of two file formats: .COM, which is an absolute image of the file to be executed and limits the program size to 65227 bytes (64 KB minus 256 bytes used by the program segment prefix minus 2 bytes initially placed on the stack); or .EXE, which allows a program of any size to be loaded and has a special header containing information about the program's entry point, stack size, and memory requirements, plus a relocation table.

The EXEMOD utility can be used to display or modify those fields of a .EXE program header that control the size of the stack segment and the amount of memory allocated to the program when MS-DOS loads the program into the transient program area for execution.

The /STACK*n* switch controls the number of bytes in the program's STACK segment by setting the initial SP to the hexadecimal value specified. The minimum paragraph allocation value is adjusted if necessary. The EXEMOD /STACK*n* switch should be used only with programs compiled by Microsoft C version 3.0 or later, Microsoft Pascal version 3.3

or later, or Microsoft FORTRAN version 3.0 or later. Use of the /STACK*n* switch with a program developed with another compiler can cause the program to fail or cause EXEMOD to return an error message.

The /MAX*n* switch specifies the maximum number of additional paragraphs of memory to allocate for use by the program. The /MIN*n* switch specifies the minimum number of paragraphs of memory, in addition to the size of the program itself and its stack and data segments, that are required for the program to execute. If enough memory exists to satisfy the minimum additional paragraphs requested but not enough exists to satisfy the maximum, MS-DOS allocates all available memory to the program.

To display the current memory allocation and stack size values from a .EXE file's header, the /H switch can be used or the file's name can be entered as the only parameter in the command line.

When EXEMOD is used on a previously packed .EXE file (a file that was processed by EXEPACK or linked with the /EXEPACK switch), the values set or displayed in the file's header are the values that will apply after the file is expanded at load time. EXEMOD displays a message advising the user that the file being modified was previously packed.

The EXEMOD switches /MAX*n* and /STACK*n* correspond to the Microsoft Object Linker's /CPARMAXALLOC:*n* and /STACK:*n* switches, respectively. *See* PROGRAMMING UTILITIES: LINK.

Return Codes

0 No error; EXEMOD operation was successful.
1 An error was encountered that terminated execution of the EXEMOD program.

Examples

To display the values in the file header of the DUMP.EXE program, type

```
C>EXEMOD DUMP.EXE  <Enter>
```

or

```
C>EXEMOD DUMP.EXE /H  <Enter>
```

The EXEMOD utility displays the following:

```
Microsoft (R) EXE File Header Utility  Version 4.02
Copyright (C) Microsoft Corp 1985.  All rights reserved.
DUMP.EXE                            (hex)           (dec)

.EXE size (bytes)                     580            1408
Minimum load size (bytes)             383             899
Overlay number                          0               0
Initial CS:IP                   0000:0000
Initial SS:SP                   0034:0040              64
Minimum allocation (para)               5               5
Maximum allocation (para)            FFFF           65535
Header size (para)                     20              32
Relocation table offset                20              32
Relocation entries                      1               1
```

To change the size of the STACK segment for the DUMP.EXE program to 400H (1024) bytes, type

```
C>EXEMOD DUMP.EXE /STACK 400  <Enter>
```

EXEMOD displays the message

```
EXEMOD : warning U4051: minimum allocation less than stack; correcting minimum
```

Messages

error U1050: usage : exemod file [-/h] [-/stack n] [-/max n] [-/min n]
An error was detected in the EXEMOD command line.

error U1051: invalid .EXE file : bad header
The file is not an executable file or has an invalid file header.

error U1052: invalid .EXE file : actual length less than reported
The file size indicated in the .EXE file header does not match the size recorded in the disk directory.

error U1053: cannot change load-high program
The header of the file cannot be modified because the minimum allocation value and the maximum allocation value are both zero.

error U1054: file not .EXE
The file specified does not have a .EXE extension.

error U1055: *filename* : cannot find file
The .EXE file specified in the command line cannot be found.

error U1056: *filename* : permission denied
The .EXE file specified in the command line is read-only.

warning U4050: packed file
The specified file is a packed file; that is, it was previously processed with the EXEPACK utility or was linked with the /EXEPACK switch. This is an informational message only; EXEMOD still modifies the file. The header values displayed are the values that will apply after the packed value is expanded at load time.

warning U4051: minimum allocation less than stack; correcting minimum
The minimum allocation value is not large enough to accommodate the stack; the minimum allocation value is adjusted. This is an informational message only.

warning U4052: minimum allocation greater than maximum; correcting maximum
If the minimum allocation value is greater than the maximum allocation value, the maximum value is adjusted. This is an informational message only.

EXEPACK

Compress .EXE File

Purpose

Compresses an executable .EXE program file so that it requires less space on the disk. The EXEPACK utility is supplied with the Microsoft Macro Assembler (MASM), C Compiler, FORTRAN Compiler, and Pascal Compiler. This documentation describes EXEPACK version 4.04.

Syntax

EXEPACK *exe_file packed_file*

where:

exe_file	is the name of the executable .EXE program file to be compressed.
packed_file	is the name of the compressed program file.

Description

The EXEPACK utility compresses an executable .EXE program by packing sequences of identical bytes and optimizing the relocation table. The EXEPACK utility is not compatible with versions of MS-DOS earlier than 2.0.

The *exe_file* parameter specifies the name of the program file produced by the Microsoft Object Linker (LINK) and must contain the extension .EXE. The *packed_file* parameter specifies the name and extension of the resulting compressed file. EXEPACK has no default extensions.

The name for *packed_file* must be different from the *exe_file* filename. Although it is possible to fool EXEPACK into creating a packed file with the same name by specifying a different but equivalent pathname for the output file, the resulting packed file will probably be damaged. If the packed file is to replace the original .EXE file, a different name should be specified for the packed file; then the input file should be deleted and the packed file renamed with the name of the original file.

When EXEPACK is used to compress an executable overlay file or a program that calls overlays, the packed file should be renamed with its original name before use to avoid interruption by the overlay-manager prompt.

The effects of EXEPACK depend on program characteristics. Most programs can be processed with EXEPACK to occupy significantly less disk space. Programs thus compressed also load for execution more quickly. Occasionally programs (particularly small ones) actually become larger after processing with EXEPACK; in such cases the file produced by EXEPACK should be discarded. Microsoft Windows programs or programs to be debugged under DEBUG, SYMDEB, or CodeView should not be compressed with EXEPACK.

Using EXEPACK on a previously linked program is equivalent to specifying LINK's /EXEPACK switch while linking that program.

Note: When using the EXEMOD utility with packed .EXE files created with EXEPACK or the /EXEPACK linker switch, use the EXEMOD version shipped with LINK or with the EXEPACK utility to ensure compatibility.

Return Codes

0 No error; the EXEPACK operation was successful.
1 An error was encountered that terminated execution of the EXEPACK utility.

Example

To compress the file BUILD.EXE into a file named BUILDX.EXE, type

```
C>EXEPACK BUILD.EXE BUILDX.EXE  <Enter>
```

Messages

fatal error U1100: out of space on output file
The destination disk has insufficient space for the output file, or the root directory is full.

fatal error U1101: *filename* : file not found
The .EXE file specified in the command line cannot be found.

fatal error U1102: *filename* : permission denied
A file with the same name as the specified output file already exists and is read-only.

fatal error U1103: cannot pack file onto itself
The file cannot be compressed because the name specified for the packed file is the same as the name of the source .EXE file.

fatal error U1104: usage : exepack <infile> <outfile>
The command line contained a syntax error, or the output filename was not specified.

fatal error U1105: invalid .EXE file; bad header
The file is not an executable file or has an invalid file header.

fatal error U1106: cannot change load-high program
The file cannot be compressed because the minimum allocation value and the maximum allocation value are both zero. *See also* PROGRAMMING UTILITIES: EXEMOD.

fatal error U1107: cannot pack already-packed file
The file specified has already been packed with EXEPACK.

fatal error U1108: invalid .EXE file; actual length less than reported
The file size indicated in the .EXE file header does not match the size recorded in the disk directory.

fatal error U1109: out of memory
The EXEPACK utility did not have enough memory to operate.

fatal error U1110: error reading relocation table
The file cannot be compressed because the relocation table cannot be found or is invalid.

fatal error U1111: file not suitable for packing
The file could not be packed because the packed load image of the specified file was larger than the unpacked load image.

fatal error U1112: *filename* : unknown error
An unknown system error occurred while the specified file was being processed.

warning U4100: omitting debug data from output file
EXEPACK has stripped all symbolic debug information from the output file.

LIB

Library Manager

Purpose

Creates or modifies an object module library file. The LIB utility is supplied with the Microsoft Macro Assembler (MASM), C Compiler, FORTRAN Compiler, and Pascal Compiler. This documentation describes LIB version 3.06.

Syntax

LIB

or

LIB *library_file* [/PAGESIZE:*n*] [*operation*][,[*list_file*][,[*new_library_file*]]] [;]

or

LIB @*response_file*

where:

library_file is the name of the object module library file to be created or modified (default extension = .LIB).

/PAGESIZE:*n* is the page size of the library file and must immediately follow *library_file* if used; *n* is a power of 2 between 16 and 32768, inclusive (default = 16). Can be abbreviated /P:*n*.

operation is one or more library manipulations to be performed. Each *operation* is specified as a code followed by an object module name (case is not significant):

+*name*	Add object module or another library to library.
−*name*	Delete object module from library.
−+*name*	Replace object module in library.
**name*	Copy object module from library to object file.
−**name*	Copy object module to object file and then delete object module from library.

list_file is the name of the file or character device to receive the cross-reference listing for the library file (default = NUL device).

new_library_file is the name to be assigned to the modified object module library file. (The default name is the same as *library_file*; if the default is used, the original *library_file* is renamed with the extension .BAK.)

response_file is the name of a text file containing LIB parameters in the same order in which they are supplied if entered interactively. The name of the response file must be preceded by the @ symbol.

Description

The Microsoft Library Manager (LIB) creates and modifies library files, checks existing library files for consistency, and prints listings of the contents of library files. The LIB utility does not work with versions of MS-DOS earlier than 2.0.

A library file consists of relocatable object modules that are indexed by their names and public symbols. The Microsoft Object Linker (LINK) uses these files during the creation of an executable (.EXE) program to resolve external references to routines and variables contained in other object modules.

The *library_file* parameter specifies the name of the object module library file to be created or modified. This parameter is required; if it is not included, LIB prompts for it. The default extension for a library file is .LIB.

The /PAGESIZE:*n* switch (abbreviated /P:*n*) sets the page size (in bytes) for a new library file or changes the page size of an existing library file. The value of *n* must be a power of 2 between 16 and 32768, inclusive. The default is 16 for a new library file; for an existing library file, the default is the current page size. Because the index to a library file is contained in a fixed number of pages, setting a larger page size increases the number of index entries (and thus the number of object modules) that a library file can contain but results in more wasted disk space (an average of half a library page per object module).

The *operation* parameter specifies one or more relocatable object modules to add to, replace in, copy from, move from, or delete from *library_file*. Each operation is represented by a code specifying the type of operation, followed by the object module name. When an object module is copied or moved from the library file, the drive and pathname of the object module are set to the default drive, current directory, and specified module name, and the extension of the object module defaults to .OBJ. When an object module is added or replaced, LIB assumes a default extension of .OBJ.

The operation +*name* adds the object module in the file *name*.OBJ to the library file. This operation can also be used to add the contents of another entire object module library file to the library file being updated, in which case the extension .LIB must be included in *name*. The operation –*name* deletes the object module *name* from the library. The operation –+*name* deletes the object module *name* from the library file and replaces it with the contents of the file *name*.OBJ. The operation **name* copies the object module *name* from the library file into the file *name*.OBJ, which LIB creates in the current directory. The operation –**name* also copies the object module *name* from the library file into a .OBJ file but then deletes the module from the library file. (Although *name* must have exactly the same spelling as the name in the library's reference listing, case is not significant.)

Note: LIB does not actually delete object modules from the specified library file. Instead, it marks the selected object modules for deletion, creates a new library file, and copies only

the modules not marked for deletion into the new file. Thus, if LIB is terminated for any reason, the original file is not lost. Enough space must be available on the disk for both the original library file and the copy.

The *list_file* parameter specifies the file or character device to receive a reference listing for the library file. Any valid drive, pathname, and extension or any valid character device, such as PRN, is permitted (default = NUL). If this parameter is omitted, no listing is generated.

The reference listing consists of two tables. The first table contains all the public symbols in the object modules in the library, listed alphabetically, with each symbol followed by the name of the object module in which it is referenced. The second table contains the names of all the object modules, listed alphabetically, with each name followed by the offset from the start of the library file, the code and data size, and an alphabetic listing of the public symbols in that object module.

The *new_library_file* parameter specifies the name for the modified library file that is created. If this parameter is omitted, LIB gives the modified library file the same name as the original library file, and the original library file is renamed with a .BAK extension. When a new library file is being created, this parameter is not necessary.

When the command line is used to supply LIB with filenames and switches, typing a semicolon character (;) after any parameter (except *library_file*) causes LIB to use the default values for the remaining parameters. If a semicolon is entered after *library_file*, LIB simply checks the file for consistency and usability. (This is seldom necessary, because LIB checks each object module for consistency before adding it to the library.)

If the LIB command is entered without any parameters, LIB prompts the user for each parameter needed. If there are too many operations to fit on one line, the line can be ended with the ampersand character (&), causing LIB to repeat the *Operations:* prompt. If any response except *library_file* is terminated with a semicolon character, LIB uses the default values for the remaining filenames. When the *library_file* parameter is followed by a semicolon or a semicolon is entered at the *Operations:* prompt, LIB takes no action except to verify that the contents of the specified file are consistent and usable.

The *response_file* parameter allows the automation of complex LIB sessions involving many files. A response file contains ASCII text that corresponds line for line to the responses that are entered in a normal interactive LIB session, in the form

library_file [/P:*n*]
[Y]
[*operations*]
[*list_file*]
[*new_library_file*] [;]

The response file name must be preceded in the command line by the at symbol (@) and can also be preceded by a path and/or drive letter. If *library_file* is a new file, the letter *Y*

must appear by itself on the second line of the response file to approve the creation of a library file. The last line of the response file must end with a semicolon or a carriage return. (LIB ignores any lines following a semicolon.) If all the parameters required by LIB are not present in the response file or the response file does not end with a semicolon, LIB prompts the user for the missing information.

Return Codes

0 No error; LIB operation was successful.
1 An error that terminated execution of the LIB utility was encountered.

Examples

To create a library file named MYLIB.LIB and insert the object files VIDEO.OBJ, COMM.OBJ, and DOSINT.OBJ, type

```
C>LIB MYLIB +VIDEO +COMM +DOSINT;  <Enter>
```

To print a listing of the object modules in the library file MYLIB.LIB, type

```
C>LIB MYLIB,PRN  <Enter>
```

If the LIB command is entered without parameters, the user is prompted for the necessary information. For example, if the user wanted to add the module VIDEO.OBJ to the library file SLIBC.LIB, produce a reference listing in the file SLIBC.LST, and produce a new output library file named SLIBC2.LIB, the following dialogue would take place:

```
C>LIB  <Enter>

Microsoft (R) Library Manager  Version 3.06
Copyright (C) Microsoft Corp 1983, 1984, 1985, 1986.  All rights reserved.

Library name: SLIBC  <Enter>
Operations: +VIDEO  <Enter>
List file: SLIBC.LST  <Enter>
Output library: SLIBC2  <Enter>
```

Messages

***filename*: cannot access file.**
LIB is unable to access an object module specified in a response file, in the command line, or at the *Operations:* prompt.

***filename*: cannot create extract file**
The object module cannot be copied or moved from the library file into a separate disk file called *filename* because the root directory or disk is full or because *filename* already exists and is read-only.

***filename*: cannot create listing**
The list file specified in the response file, in the command line, or at the *List file:* prompt cannot be created because the root directory or disk is full or because *filename* already exists and is read-only.

***filename*: invalid format (*xxxx*); file ignored.**
The hexadecimal signature byte or word *xxxx* of the specified file was not one of the following recognized types: Microsoft library, Intel library, Microsoft object, or XENIX archive.

***filename*: invalid library header.**
The input library file either is not a library file or is damaged.

***filename*: invalid library header; file ignored.**
The input library file is in the wrong format.

modulename*: invalid object module near *location
The specified object module has an invalid format near the hexadecimal offset indicated.

***modulename*: module not in library; ignored**
The object module specified in the response file, in the command line, or at the *Operations:* prompt is not in the specified input library file.

***modulename*: module redefinition ignored**
An object module was specified to be added to a library file but an object module with the same name was already in the library file, or the same object module was specified twice in an add operation in the command line.

***number*: page size too small; ignored**
The size specified with a /P:*n* switch must be a power of 2 between 16 and 32768 bytes, inclusive.

***symbol* (*modulename*) : symbol redefinition ignored**
The specified symbol was defined in more than one module. Only the first definition of a symbol is accepted. All redefinitions are ignored.

cannot create new library
The root directory is full, or a library file with the same name already exists and is read-only.

cannot open response file
The specified response file cannot be found or does not exist.

cannot rename old library
The old library file cannot be renamed with a .BAK extension because such a file already exists and is read-only.

cannot reopen library
The old library file could not be reopened after it was renamed with the .BAK extension. This error usually indicates damage to the operating system or to the disk directory structure.

comma or new line missing
A comma or carriage return was expected in the command line but was not found.

Do not change diskette in drive *X*:
LIB may have placed important temporary files on the specified disk. Do not remove the disk until the LIB operation is complete or these files may be lost.

error writing to cross-reference file
The disk or root directory is full.

error writing to new library
The new library file cannot be created because the disk is full.

free: not allocated
This is a serious problem. Note the circumstances of the failure and notify Microsoft Corporation.

insufficient memory
Not enough memory is available in the transient program area for LIB to successfully perform the requested operations.

internal failure
This is a serious problem. Note the circumstances of the failure and notify Microsoft Corporation.

Library does not exist. Create?
The specified *library_file* does not exist on disk. Respond with *Y* to create the library file; respond with *N* to terminate the LIB utility.

mark: not allocated
This is a serious problem. Note the circumstances of the failure and notify Microsoft Corporation.

option unknown
The command line included a switch other than /P:*n*.

output-library specification ignored
An output library file was specified in addition to a new library file. This is only a warning. The output library file specification will be disregarded.

page size too small
The page size of an input library file was less than 16 bytes, indicating a damaged or otherwise invalid .LIB file. *See* LIB message *number*: page size too small; ignored.

syntax error
The command line included an invalid parameter or switch.

syntax error: illegal file specification
A command operator (such as *, –, or +) was given without an object module name.

syntax error: illegal input
The command line included an invalid parameter or switch.

syntax error: option name missing
The command line included a forward slash (/) that was not followed by P:*n*.

syntax error: option value missing
The /P switch was not followed by the page size value in bytes.

terminator missing
Either a control character (such as Control-Z) was specified at the *Output library:* prompt or the response file line that corresponds to LIB's *Output library:* prompt was not terminated by a carriage return or semicolon.

too many symbols
The maximum number of public symbols allowed in a library file has been exceeded. The limit for all object modules (combined) is 4609.

unexpected end-of-file on command input
The response file did not include all the necessary LIB parameters.

write to extract file failed
The destination disk has insufficient space for the complete object module, or the root directory is full.

write to library file failed
The destination disk has insufficient space to create the new library file, or the root directory is full.

LINK

Create .EXE File

Purpose

Combines relocatable object modules into an executable (.EXE) file. The Microsoft Object Linker (LINK) is supplied with the Microsoft Macro Assembler (MASM), C Compiler, Pascal Compiler, and FORTRAN Compiler. This documentation describes LINK version 3.50.

Syntax

LINK

or

LINK *obj_file*[+*obj_file*...][,[*exe_file*]][,[*map_file*]][,[*library*[+*library*...]]] [*options*] [;]

or

LINK @*response_file*

where:

obj_file is the name of a file containing a relocatable object module produced by MASM or by a high-level-language compiler (default extension = .OBJ).

exe_file is the name of the executable file to be produced by LINK (default extension = .EXE).

map_file is the name of the file or character device to receive a listing of the names, load addresses, and lengths of the segments in *exe_file* (default = NUL device; default extension = .MAP).

library is the name of an object module library to be searched to resolve external references in the object file(s) (default extension = .LIB).

response_file is the name of a text file containing LINK parameters in the order in which they are supplied during an interactive LINK session.

options specifies one or more of the following switches. Switches can be either uppercase or lowercase.

- /CP:*n* (/CPARMAXALLOC:*n*) Sets the maximum number of extra memory paragraphs required by *exe_file* (default = 65535).
- /DS (/DSALLOCATE) Loads the data in DGROUP at the high end of the data segment.
- /DO (/DOSSEG) Arranges segments according to the Microsoft language segment-ordering convention.
- /E (/EXEPACK) Compresses repetitive sequences of bytes and optimizes *exe_file*'s relocation table.

(more)

/HI	(/HIGH) Causes *exe_file* to be loaded as high as possible in memory when *exe_file* is executed.
/HE	(/HELP) Lists LINK options on the screen. No other switches or filenames should be used with this switch.
/LI	(/LINENUMBERS) Copies line-number information (if available) from *obj_file* to *map_file*. If a map file was not specified, this switch creates one.
/M	(/MAP) Copies a list of all public symbols declared in *obj_file* to *map_file*. If a map file was not specified, this switch creates one.
/NOD	(/NODEFAULTLIBRARYSEARCH) Causes LINK to ignore any library names inserted in the object file by the language compiler.
/NOG	(/NOGROUPASSOCIATION) Causes LINK to ignore GROUP associations when assigning addresses.
/NOI	(/NOIGNORECASE) Causes LINK to be case sensitive when resolving external names.
/O:*n*	(/OVERLAYINTERRUPT:*n*) Overrides the interrupt number used by the overlay manager (0–255, default = 63, or 3FH). This switch should be used only when linking with a run-time module from a language compiler that supports overlays.
/P	(/PAUSE) Causes LINK to pause and prompt the user to change disks before writing the *exe_file*.
/SE:*n*	(/SEGMENTS:*n*) Sets the maximum number of segments that can be processed (1–1024, default = 128).
/ST:*n*	(/STACK:*n*) Sets the size of the *exe_file*'s stack segment to *n* bytes (1–65535).

Description

LINK combines relocatable object modules into an executable file in the .EXE format. LINK can be used with object files produced by any high-level-language compiler or assembler that supports the Microsoft object module format. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING TOOLS: Object Modules; The Microsoft Object Linker.

The *obj_file* parameter, which is required, specifies one or more files containing relocatable object modules. If multiple object files are linked, their names should be separated by a plus operator (+) or a space. If an extension is not specified for an object file, LINK supplies the extension .OBJ. Some high-level-language compilers support partitioning of the executable program into a root segment and one or more overlay segments and include a special overlay manager in their libraries; when these compilers are used, the object modules that compose each overlay segment should be surrounded with parentheses in the LINK command line.

The *exe_file* parameter specifies the name of the executable file that is created by LINK. The default is the same filename as the first object file, but with the extension .EXE.

The *map_file* parameter designates the file or character device to receive LINK's listing of the name, load address, and length of each of *exe_file*'s segments. The map file also includes the names and load addresses of any groups in the program, the program entry point, and, if the /M switch is used, all public symbols and their addresses. If the /LI switch is used and if line numbers were inserted into *obj_file* by the compiler, the starting address of each *obj_file* program line is also copied to *map_file*. The default extension for a map file is .MAP. If the /M or /LI switch is used, a map file is created using the name of the specified .EXE file even if *map_file* is not specified. If neither the /M nor the /LI switch is used and *map_file* is not specified, no listing is created.

The *library* parameter specifies the object module library or libraries that will be searched to resolve external references after all the object files are processed. The default extension for library files is .LIB. Multiple library names should be separated by plus operators (+) or spaces. A maximum of 16 search paths can be specified in the LINK command line. If a library name is preceded by a drive and/or path, LINK searches only the specified location. If no drive or path precedes a library name, LINK searches for library files in the following order:

1. Current drive and directory
2. Any other library search paths specified in the command line, in the order they were entered
3. Directories specified in the LIB= environment variable, if one exists

In the following example, LINK searches only the \ALTLIB directory on drive A to find the library MATH.LIB. To find the library COMMON.LIB, LINK searches the current directory on the current drive, then the current directory on drive B, then directory \LIB on drive D, and finally, any directories named in the LIB environment variable.

```
C>LINK TEST,,TEST,A:\ALTLIB\MATH.LIB+COMMON+B:+D:\LIB\  <Enter>
```

If default libraries are specified within the object files through special records inserted by certain high-level-language compilers, those libraries will be searched *after* the libraries named in the command line or response file.

If the LINK command is entered without parameters, LINK prompts the user for each filename needed. The default response for each prompt (except the *obj_file* prompt) is displayed in square brackets and can be selected by pressing the Enter key. If there are too many *obj_file* or *library* names to fit on one line, the line can be terminated by entering a plus operator (+) and pressing the Enter key; LINK then repeats the prompt. If the user ends any response with a semicolon character (;), LINK uses the default values for the remaining fields.

When the command line contains filenames and switches, commas must be used to separate the *obj_file*, *exe_file*, *map_file*, and *library* parameters. If a filename is not supplied, a comma must be used to mark its place. If the user places a semicolon after any parameter in the command line, LINK terminates the command line at the semicolon and uses the default values for any remaining parameters.

The user can automate complex LINK sessions involving multiple files by creating a response file. The *response_file* parameter must be the name of an ASCII file that corresponds line for line to the responses that are entered in a normal interactive LINK session. The last line of the response file must end with a semicolon character (;) or a carriage return. If all parameters required by LINK are not present in the response file and the response file does not end with a semicolon or carriage return, LINK prompts the user for the missing information.

LINK supports many options that can be invoked by including a switch in the command line, as part of the response to a LINK prompt, or in a response file. To simplify this description, these switches are grouped according to their functions.

The /E, /HE, /NOD, /NOI, /P, and /SE:*n* switches affect LINK's general operation. The /E switch compresses repetitive sequences of bytes in *exe_file* and optimizes certain parts of the relocation table in *exe_file*'s header. The /E switch functions exactly like the EXEPACK utility.

Note: The /E switch does not always save a significant amount of disk space and may even increase file size when used with small programs that have few load-time relocations or repeated characters. The Microsoft Symbolic Debugger (SYMDEB) utility cannot be used with packed files.

The /HE switch displays the available options on the screen. No other switches or filenames should be specified if the /HE switch is used. The /NOD switch causes LINK to ignore any default libraries that have been added to the object modules by the high-level-language compiler that produced the modules, thus restricting searches to those libraries specified in the command line or response file. The /NOI switch causes LINK to be case sensitive when resolving external references to symbols between object modules. The /NOI switch is typically used with object files created by high-level-language compilers that differentiate between uppercase and lowercase letters.

The /P switch causes LINK to pause and prompt the user before writing *exe_file* to disk, thus allowing the user to exchange the disk used during the linking operation for another that has more space available. The /SE:*n* switch controls the number of program segments processed by LINK. The *n* must be a decimal, octal, or hexadecimal number from 1 through 1024, inclusive (default = 128). Octal numbers must have a leading zero; hexadecimal numbers must begin with 0x.

The /M and /LI switches affect the production and contents of the optional map file. The /M switch creates a map file with the same name as *exe_file* or, if *exe_file* is not specified, with the same name as the first object file and the extension .MAP. The resulting map file includes a list of all public symbols and their addresses. The /LI switch also creates a map file and includes line-number information if available in the object file. (MASM and some high-level-language compilers do not insert line-number information into object files.)

The /D, /DO, /NOG, and /O:*n* switches affect the structure of the code in *exe_file*. Use of the /D switch places the data in DGROUP at the top (highest address) of the memory segment pointed to by the DS register, rather than at the bottom (the default). The /DO switch arranges the program segments according to a convention expected by all Microsoft language compilers: All segments with the class name CODE are placed first in the executable file; any other segments that do not belong to DGROUP are placed immediately after the CODE segments; all segments belonging to DGROUP are placed at the end of the file. The /NOG switch causes LINK to ignore group associations specified in the object modules when assigning addresses to data and code items; that is, segments that would ordinarily have been collected into the same physical memory segment because of their association within a GROUP are decoupled. The /NOG switch provides compatibility with LINK versions 2.02 and earlier and with early versions of Microsoft language compilers. The /O:*n* switch controls the interrupt number used by the resident overlay manager if the linked program includes overlays. The number *n* can be any decimal, octal, or hexadecimal number in the range 0 through 255 (default = 63, or 3FH). Octal numbers must have a leading zero; hexadecimal numbers must begin with 0x.

Note: MASM and many high-level-language compilers do not include overlay managers in their libraries. Users should check their compiler documentation to determine if the /O:*n* switch can be used.

Warning: Interrupt numbers that conflict with the software interrupts used to obtain MS-DOS or ROM BIOS services or with hardware interrupts assigned to peripheral device controllers should not be used in the /O:*n* switch.

The /C:*n*, /H, and /ST:*n* switches control the information in *exe_file*'s header that affects the behavior of the MS-DOS system loader when the file is read from the disk into RAM for execution. The /C:*n* switch sets the maximum number of 16-byte paragraphs of memory to be made available to the program when it is loaded into memory, in addition to the memory required to hold the program's code, data, and stacks; the default is 65535, which causes the program to be allocated all available memory. The /H switch causes the program to be loaded as high as possible in the transient program area (free memory), rather than as low as possible (the default). The /ST:*n* switch sets the stack size (in bytes) to be allocated for the program when it is loaded and overrides any stack segment size declarations in the original source code. The number *n* can be any decimal, octal, or hexadecimal number from 1 through 65535; however *n* must be large enough to accommodate any initialized data in the stack segment. Octal numbers must have a leading zero; hexadecimal numbers must begin with 0x. If the /ST:*n* switch is not used, LINK calculates a program's stack size, basing the size on the size of any stack segments given in the object files. The /C:*n* and /ST:*n* values in the *exe_file* header can be altered after linking by using the EXEMOD utility.

If LINK is unable to hold in RAM all the data it is processing, it creates a temporary disk file named VM.TMP (Virtual Memory) in the current directory of the default disk drive. If a floppy disk is in the default drive, LINK issues a warning message to prevent the user from changing disks until the LINK session is completed. After LINK finishes processing, it deletes the temporary file.

Warning: Any file named VM.TMP that is already on the disk will be destroyed if LINK creates the temporary disk file.

Return Codes

0 No errors or unresolved references were encountered during creation of *exe_file*.
1 A miscellaneous LINK error occurred that was not covered by the other return codes.
16 A data record was too large to process.
32 No object files were specified in the command line or response file.
33 The map file could not be created.
66 A COMMON area was declared that is larger than 65535 (one segment).
96 Too many libraries were specified.
144 An invalid object module (*obj_file*) was detected.
145 Too many TYPDEFs were found in the specified object modules.
146 Too many group, segment, or class names were found in one object module.
147 Too many segments were found in all the object modules combined, or too many segments were found in one object module.
148 Too many overlays were specified.
149 The size of a segment exceeded 65535.
150 Too many groups or GRPDEFs were found in one object module.
151 Too many external symbols were found in one object module.
177 The size of a group exceeded 65535.

Examples

The simplest use of LINK is to process a single object file to produce an executable file, using all the default values. For example, to process the file SHELL.OBJ, create an executable file named SHELL.EXE, and search only the default libraries, type

```
C>LINK SHELL;  <Enter>
```

The semicolon after the filename causes LINK to use the default values for all other parameters.

To link three object files named SHELL.OBJ, VIDEO.OBJ, and DOSINT.OBJ into an executable file named SHELL.EXE and search the library DEVLIB.LIB on drive B before searching any default libraries, type

```
C>LINK SHELL+VIDEO+DOSINT,,,B:DEVLIB  <Enter>
```

If the LINK command is entered without parameters, LINK prompts the user for the necessary information. For example, the following interactive session links the file

MENUMGR.OBJ into the executable file MENUMGR.EXE, creates a map file named MENUMGR.MAP, and searches the math floating-point emulator library EM.LIB before any default libraries:

```
C>LINK  <Enter>

Microsoft (R) 8086 Object Linker  Version 3.05
Copyright (C) Microsoft Corp 1983,1984,1985.  All rights reserved.

Object Modules [.OBJ]: MENUMGR  <Enter>
Run File [MENUMGR.EXE]:  <Enter>
List File [NUL.MAP]:  MENUMGR  <Enter>
Libraries [.LIB]:  EM  <Enter>
```

Messages

***filename* is not a valid library**
The file specified as an object module library either is corrupt or is not a library in the format created by the Microsoft LIB utility.

About to generate .EXE file
Change diskette in drive *X* and press <ENTER>
The /P switch was used in the command line. LINK is prompting the user to change disks before LINK creates the file containing the executable program.

Ambiguous switch error: "*option*"
A valid switch was not entered after a forward slash (/) in the command line.

Array element size mismatch
A FAR communal array was declared with two or more different array-element sizes (for example, once as an array of characters and once as an array of real numbers). This error occurs only with programs produced by the Microsoft C Compiler or other compilers that support FAR communal arrays; it does not occur with object files produced by MASM.

Attempt to access data outside segment bounds
A data record in an object module specified data extending beyond the end of a segment. This is a translator error. Note which compiler or assembler produced the invalid object module and notify Microsoft Corporation.

Attempt to put segment *name* in more than one group in file *filename*
A segment was declared to be a member of two groups. Correct the source code and re-create the object modules.

Bad value for cparMaxAlloc
The value specified using the /C:*n* option is not in the range 1 through 65535.

Cannot create temporary file
The destination disk has insufficient space for the temporary file, or the root directory is full.

Cannot find file *filename*
Change diskette and press <ENTER>
The specified object file cannot be found in the current drive.

Cannot find library: *filename*
Enter new file spec:
The specified library file cannot be found or does not exist. Enter the correct drive letter, check the spelling of the filename and path, or make sure that the LIB environment variable has been set up properly.

Cannot nest response files
A response file was named within a response file. Revise the response file to eliminate the nested file.

Cannot open list file
The destination disk has insufficient space for the listing, or the root directory is full.

Cannot open response file: *filename*
LINK cannot find the specified response file.

Cannot open run file
The destination disk has insufficient space for the .EXE file, or the root directory is full.

Cannot open temporary file
The destination disk has insufficient space for the temporary file, or the root directory is full.

Cannot reopen list file
The original disk was not replaced when requested. Restart LINK.

Common area longer than 65536 bytes
The program has more than 64 KB of communal variables. This error occurs only with programs produced by the Microsoft C Compiler or other compilers that support communal variables.

Data record too large
An LEDATA record (in an object module) contains more than 1024 bytes of data. This is a symptom of an error in the compiler used to generate the object module. Document the circumstances and contact Microsoft Corporation.

Dup record too large
An LIDATA record (in an object module) contains more than 512 bytes of data. This error may be caused by a complex structure definition or by a series of deeply nested DUP operators.

File not suitable for /EXEPACK, relink without
The file linked with the /E switch would have been smaller if it had not been compressed. Relink without the /E switch.

Fixup overflow near *number* in segment *name* in *filename* offset *number*
A group is larger than 64 KB, the original source file contains an intersegment short jump or intersegment short call, the name of a data item conflicts with that of a library subroutine, or an EXTRN declaration is placed inside the wrong segment.

Incorrect DOS version, use DOS 2.0 or later
LINK uses the extended file management calls to provide path support and, thus, does not work with versions of MS-DOS earlier than 2.0.

Insufficient stack space
Not enough memory is available to run LINK.

Interrupt number exceeds 255
The number specified in the /O:*n* switch is not in the range 0 through 255.

Invalid numeric switch specification
An incorrect value was entered with one of the LINK options.

Invalid object module
One of the object modules is invalid. Recompile the source file. If the error persists after recompiling, document the circumstances and contact Microsoft Corporation.

NEAR/HUGE conflict
Conflicting NEAR and HUGE definitions were given for a communal variable. This error occurs only with programs produced by the Microsoft C Compiler or other compilers that support communal variables.

Nested left parentheses
An opening (left) parenthesis is needed on the left side of an overlay module.

Nested right parentheses
A closing (right) parenthesis is needed on the right side of an overlay module.

No object modules specified
No object file names were specified in the command line or response file.

Object not found
One of the object files specified in the command line was not found.

Out of space on list file
The destination disk has insufficient space for the listing.

Out of space on run file
The destination disk has insufficient space for the .EXE file.

Out of space on scratch file
The disk in the default drive has insufficient space for temporary files.

Overlay manager symbol already defined: *name*
A symbol name was defined that conflicts with one of the special overlay manager names. Use another symbol name.

Please replace original diskette in drive *X* and press <ENTER>
The /P switch was specified in the command line and the disk to receive the .EXE file produced by LINK has already been inserted. This message indicates that the .EXE file was successfully created and that the original disk should again be placed in the drive.

Relocation table overflow
More than 32768 long calls, long jumps, or other long pointers were found in the program. The program may need to be restructured to reduce the number of FAR references. (Pascal and FORTRAN users should try turning off the debugging option before restructuring the program.)

Response line too long
A line in a response file had more than 127 characters.

Segment limit set too high
The number specified in the /SE:*n* switch was not in the range 1 through 1024.

Segment limit too high
Not enough memory is available for LINK to allocate tables to describe the number of segments requested (default = 128 or the number specified in the /SE:*n* switch). Use the /SE:*n* switch to specify a smaller number of segments, or alter the system configuration to increase the amount of free memory.

Segment size exceeds 64K
The program is a small-model program with more than 64 KB of code or data, a compact-model program with more than 64 KB of code, or a medium-model program with more than 64 KB of data. Selection of a different model or alteration of the program code may be required to successfully complete the LINK process.

Stack size exceeds 65536 bytes
The size specified for the stack in the /ST:*n* switch was too large, or the combined length of multiple declared stack segments exceeded 64 KB.

Symbol already defined: "*symbol*"
One of the special overlay symbols required for overlay support was previously defined.

Symbol defined more than once: "*symbol*" in file
A symbol has been defined more than once in the object module. Remove the extra symbol definition.

Symbol table overflow
The program has more than 256 KB of symbolic information (publics, externals, segments, groups, classes, files, and so on). Eliminate as many public symbols as possible, combine modules and/or segments, and recreate the object files.

Terminated by user
Ctrl-C or Ctrl-Break was pressed, causing the LINK session to be terminated prematurely.

Too many external symbols in one module
An object module contains more than the limit of 1023 external symbols.

Too many group-, segment-, and class-names in one module
One of the object modules for the program contains too many group, segment, and class names. The source file for the object module may need to be divided or restructured.

Too many groups
The program defines more than nine groups (including DGROUP). Groups must be combined or eliminated.

Too many GRPDEFs in one module
LINK encountered more than nine group definitions (GRPDEFs) in a single object module. Reduce the number of GRPDEFs or split the object module.

Too many libraries
More than 16 libraries were specified. Combine libraries or use object modules that require fewer libraries.

Too many overlays
The program defines more than 63 overlays. Reduce the number of overlays.

Too many segments
The program has more than the maximum number of segments as specified by the default of 128 or with the /SE:*n* switch. Use the /SE:*n* switch to specify a greater number of segments.

Too many segments in one module
An object module has more than 255 segments. Split the module or combine segments.

Too many TYPDEFs
An object module contains too many TYPDEF records (these records describe communal variables). This error occurs only with programs produced with the Microsoft C Compiler or other compilers that support communal variables.

Unexpected end-of-file on library
This message may indicate that the disk containing the library in use was removed prematurely.

Unexpected end-of-file on scratch file
The disk containing VM.TMP was removed.

Unmatched left parenthesis
A syntax error was detected in the specification of an overlay structure. Refer to the language compiler manual for instructions on specifying overlays to LINK.

Unmatched right parenthesis
A syntax error was detected in the specification of an overlay structure. Refer to the language compiler manual for instructions on specifying overlays to LINK.

Unrecognized switch error: *"option"*
An unrecognized character was entered after a forward slash (/) in the command line.

Unresolved COMDEF; Microsoft internal error
This is a serious problem. Note the circumstances of the failure and contact Microsoft Corporation.

Unresolved externals: *list*
A symbol was declared external (EXTRN) in one object module but was not declared PUBLIC in the object module in which it was defined, or a necessary library specification was omitted from the command line or response file.

VM.TMP is an illegal file name
and has been ignored
VM.TMP was specified as an object file name. If an object file named VM.TMP exists, rename it.

Warning: load-high disables exepack
The /H and /E switches cannot be used at the same time.

Warning: no stack segment
The program contains no segment with the STACK combine type. This message can be ignored if there is a specific reason for not defining a stack (for example, if the .EXE file will subsequently be converted to a .COM file) or for defining one without the STACK combine type.

WARNING: Segment longer than reliable size
Although code segments can be as long as 65536 bytes, code segments longer than 65500 bytes can be unreliable on the Intel 80286 microprocessor. Reduce all code segments to 65500 bytes or less.

Warning: too many public symbols
The /M switch was used to request a sorted listing of public symbols in the map file, but there are too many symbols to sort. LINK will produce an unsorted listing instead.

MAKE

Maintain Programs

Purpose

Interprets a text file of commands to compare dates of files and carry out other operations on the basis of the comparison. MAKE is customarily used to update the executable version of a program after a change to one or more of its source files. The MAKE utility is supplied with the Microsoft Macro Assembler (MASM), C Compiler, and FORTRAN Compiler. This documentation describes MAKE version 4.05.

Syntax

MAKE [/D] [/I] [/N] [/S] [*name=value* ...] *filename*

where:

filename	is an ASCII text file that contains MAKE dependency statements, commands, macro definitions, and inference rules.
name=value	declares a MAKE macro, associating a specific value with the dummy parameter *name*.
/D	displays the last modification date of each file as it is scanned.
/I	causes MAKE to ignore exit codes returned by programs called by *filename*.
/N	displays but does not execute the commands in *filename*.
/S	selects "silent" mode (commands are not displayed as they are executed).

Note: Switches can be either uppercase or lowercase and can be preceded by a dash (-) instead of a forward slash (/). Versions of MAKE earlier than 4.0 have no switches.

Description

The MAKE utility allows maintenance of complex programs to be automated. Its basic operation is to compare the dates of files and to carry out, or not carry out, an associated list of commands on the basis of the comparison.

The *filename* parameter specifies an ASCII text file often referred to as a make file. By convention, *filename* is the same as the name of the executable program being maintained, but without an extension. A make file can contain the following types of entries:

- Dependency statements
- Commands
- Macro definitions
- Inference rules
- Comments

The basic form of a make file is a dependency statement followed by one or more valid MS-DOS command lines:

targetfile: *dependentfile1* [*dependentfile2*...]
    command1
    [*command2*]
    ...

where *targetfile* designates the file that may need updating, *dependentfile* is a source file or files on which *targetfile* depends, and *command1*, *command2*, and so forth are any valid MS-DOS internal commands or external programs. These commands or programs are executed only if the date and time stamps of any dependent file are more recent than those of the target file or if the target file does not exist. Only one target file can be specified. Any number of dependent files can be included; each dependent filename must be separated from the next by at least one space. If too many dependent files are included to fit on a single line, the line can be terminated with a backslash character (\) and the list continued on the next line.

Any number of MS-DOS command lines can follow a dependency statement. The last command line should be followed by a blank line to set it off from the next MAKE entry. It is recommended that each command line include a leading space or tab character for compatibility with future versions of MAKE and existing versions of XENIX MAKE.

A macro definition takes the form

name=value

where both *name* and *value* are any string. Whenever *name* is referenced in the make file in the form $(*name*), *name* is replaced by the string *value* before the statement that contains it is evaluated or executed. Macro definitions can be nested, although very complex macro definitions can result in the premature termination of the MAKE process because of lack of memory. If *name* is not defined in the file but is defined in the system environment block by a previous SET command, $(*name*) is replaced by the string following the equal sign (=) in the environment block. If the command line contains a parameter of the form *name=value*, the command line overrides any definition of *name* in the make file or in the environment block. Thus, the precedence for macro definitions with the same *name* is

1. Command line
2. Make file
3. Environment block

MAKE contains several special macros that make it more convenient to form commands:

Macro	Action
$*	Substitutes as the base portion of *targetfile* (the filename without the extension).
$@	Substitutes as the complete *targetfile* name.
$**	Substitutes as the complete *dependentfile* list.

An inference rule specifies a series of commands to be carried out for a matching dependency statement that is not followed by its own list of commands. Inference rules allow a set of commands to be applied to more than one *targetfile: dependentfile* description, eliminating repetition of the same set of commands for several descriptions. An inference rule takes the form

.dependentextension.targetextension:
    command1
    [*command2*]
    . . .

Whenever MAKE finds a dependency statement not followed by any commands, the utility first searches the make file for an inference rule. If MAKE doesn't find an inference rule in the make file, the utility then searches the current drive and directory (or any directories specified with the MS-DOS PATH command) for the tools initialization file (TOOLS.INI) and searches the *[make]* section of TOOLS.INI for an inference rule that matches the extensions of the target file and dependent files in the dependency statement.

A make file can contain any number of comment lines. If a comment is placed where MAKE expects to find a command, the comment must be on a separate line and must have the pound character (#) as the first character of the line. Elsewhere, a pound character and following comment text can be placed either on a line alone or after the last dependent file or command listed on a line. Characters appearing on a line after the pound character are ignored during execution.

The /D, /N, and /S switches affect MAKE's output to the display while MAKE is executing. The /D switch causes the last modification date of each file to be displayed as the file is scanned. The /N switch causes the commands in the make file to be expanded and displayed, but not executed; this is useful for determining the result of a specific MAKE process without first examining the file dates and without recompiling or relinking files. The /S switch selects "silent" mode, in which commands are not displayed as they are executed.

The /I switch causes MAKE to ignore error codes returned by the compilers, assemblers, linkers, or other programs called by the make file. When the /I switch is used, the MAKE process proceeds to completion regardless of errors instead of terminating immediately as it ordinarily would, but the resulting files may not be executable.

Return Codes

0 No error; the MAKE process was successful.
1 Processing was terminated because of a fatal error by MAKE or by one of the programs called by MAKE.

Example

Assume that the file SHELL contains the following MAKE dependency statements and commands:

```
video.obj: video.asm
        masm video;

shell.obj: shell.c
        msc shell;

shell.exe: shell.obj video.obj
        link /map shell+video,shell,shell,slibc2
```

The SHELL file asserts that the executable program SHELL.EXE is composed of the files SHELL.OBJ and VIDEO.OBJ, which are in turn compiled or assembled from the source files SHELL.C and VIDEO.ASM. To update the file SHELL.EXE if either of the source files for its constituent modules has been changed, type

```
C>MAKE SHELL  <Enter>
```

Messages

fatal error U1001: macro definition larger than 512
A single macro was defined to have a value string longer than the 512-byte maximum. Rewrite the make file to use two or more short lines instead of one long line.

fatal error U1002: infinitely recursive macro
The macros defined in the make file form a circular chain.

fatal error U1003: out of memory
The make file cannot be processed because insufficent memory is available in the transient program area. Split the make file into two make files or reconfigure the system to increase available memory.

fatal error U1004: syntax error : macro name missing
A macro name is missing from the left side of the equal sign (=).

fatal error U1005: syntax error : colon missing
A line that should be a dependency statement lacks the colon that separates a target file from its dependent files. MAKE expects any line that follows a blank line to be a dependency statement.

fatal error U1006: *targetname* : macro expansion larger than 512
A single macro expansion, plus the length of any string to which it may be concatenated, is longer than 512 bytes. Rewrite the make file to use two or more short lines instead of one long line.

fatal error U1007: multiple sources
An inference rule has been defined more than once in the make file.

fatal error U1008: *filename* : cannot find file
The specified file does not exist.

fatal error U1009: *command* : argument list too long
A command line in the make file is longer than 128 characters (the maximum MS-DOS allows).

fatal error U1010: *filename* : permission denied
The specified file is read-only.

fatal error U1011: not enough memory
Memory is insufficient in the transient program area to execute a program listed in the make file. Reconfigure the system to increase available memory, if necessary.

fatal error U1012: *filename* : unknown error
This is a serious problem. Note the circumstances of the failure and notify Microsoft Corporation.

fatal error U1013: *command* : error *returncode*
One of the programs or commands called by MAKE was not able to execute correctly. MAKE terminates and displays the error code from the program that failed.

warning U4000: *filename* : target does not exist
The target file does not already exist. The dependency statement is evaluated as though the target file exists and has a date earlier than that of any of the dependent files.

warning U4001: dependent *filename* does not exist;
target *filename* not built
One of the dependent files does not exist or could not be found, so MAKE terminated without creating a new target file.

warning U4013: *command* : error *returncode* (ignored)
One of the programs or commands called by MAKE did not execute successfully and has returned the specified return code. Because MAKE was run with the /I switch, MAKE ignores the error and continues processing the make file.

warning U4014: usage : make [/n] [/d] [/i] [/s] [name=value ...] file
An error was detected in the MAKE command line.

MAPSYM

Create Symbol File for SYMDEB

Purpose

Processes a map file generated by the Microsoft Object Linker (LINK) to create a special symbol file for use with SYMDEB, the symbolic debugging program. The MAPSYM utility is supplied with the Microsoft Macro Assembler (MASM). This documentation describes MAPSYM version 4.0.

Syntax

MAPSYM [/L] *map_file*

where:

map_file	is a map file produced by LINK (default extension = .MAP).
/L	causes information about the symbol file to be displayed as it is created.

Note: The /L switch can be either uppercase or lowercase and can be preceded by a dash (-) instead of a forward slash (/).

Description

LINK combines relocatable object records (produced by MASM or a high-level-language compiler) into an executable program, which is stored in a specially formatted file with a .EXE extension. LINK can also produce an optional map file that contains information about public symbols and addresses in the linked program. The map file is an ordinary ASCII text file and has a default extension of .MAP.

To create a map file to use with MAPSYM, the LINK command line should include the /MAP switch, which creates the file, and the /LINENUMBERS switch, which includes line numbers. *See* PROGRAMMING UTILITIES: LINK.

The MAPSYM utility processes a map file into a special symbol file that can be used by SYMDEB. A drive and pathname can be specified if the map file is not in the current directory. If a file extension is not specified, .MAP is assumed.

The symbol file created by MAPSYM is placed in the current directory and has the same name as the map file but has the extension .SYM. It can contain a maximum of 1024 segments (or as many segments as can fit into available memory) and 10,000 symbols per segment. *See* PROGRAMMING UTILITIES: SYMDEB.

When the /L switch precedes *map_file* in the command line, MAPSYM displays the names of groups defined in the program described by the map and symbol files, plus the program's starting address. The /L switch does not affect the format of the symbol file that is generated.

Return Codes

0 No error; the MAPSYM process was successful.

1 Processing was terminated because of a write failure, because the map file specified does not exist, or because the symbol file could not be created.

4 Processing was terminated because an unexpected end-of-file mark was detected, because too many segments exist in the map file, because no public symbols exist in the map file, or because not enough memory is available to create the symbol file.

Example

To convert the file HELLO.MAP, which was produced by assembling and linking the file HELLO.ASM, to a symbol file that can be used by SYMDEB, type

```
C>MAPSYM /L HELLO  <Enter>
```

MAPSYM displays the following:

```
Microsoft (R) Symbol File Generator  Version 4.00
Copyright (C) Microsoft Corp 1984, 1985.  All rights reserved.
Building: HELLO.SYM
HELLO.MAP
        Program entry point at 0000:0100
HELLO      0 segment
```

The symbol file produced by MAPSYM symbol has the name HELLO.SYM.

Messages

Can't create: <*filename*>
The drive specified does not exist, the current disk or directory is full, or the output file already exists and is read-only.

Can't open MAP file: <*filename*>
The file named in the command line does not exist.

DOS 2.0 or later required
MAPSYM does not work with versions of MS-DOS earlier than 2.0.

mapsym: out of memory
System memory is insufficient to process the map file.

mapsym: segment table (*n*) exceeded.
More than 1024 segments have been used in the map file. The number displayed is the total number of segments in the map file.

No public symbols
Re-link file with the /M switch!
The map file created by LINK does not include a list of public names. The .EXE file must be relinked using the /MAP switch to generate a map file that can be used with MAPSYM.

Unexpected eof reading: <*filename*>
The map file contains no symbols, is corrupt, or is otherwise invalid. The .EXE file must be relinked and a new map file generated.

usage: MAPSYM [/l] maplist
A syntax error was detected in the command line.

Write fail on: <*filename*>
An error occurred during the creation of the output file.

MASM

Microsoft Macro Assembler

Purpose

Translates an assembly-language source program into a relocatable object module. MASM is part of the Microsoft Macro Assembler (MASM) retail package. This documentation describes MASM version 4.0.

Syntax

MASM

or

MASM *source_file* [,[*object_file*][,[*list_file*][,[*cref_file*]]]] [*options*] [;]

where:

source_file is the name of the file containing the assembly-language source code (default extension = .ASM).

object_file is the name of the file to receive the assembled object module (default extension = .OBJ).

list_file is the name of the file or device to receive the assembly listing (default = NUL). (If destination = file, default extension = .LST.)

cref_file is the name of the cross-reference file to receive information for later processing by the CREF utility (default = NUL). (If destination = file, default extension = .CRF.)

options is one or more switches from the list below.

/A	Writes the program segments in alphabetic order.
/B*n*	Sets the size of the source-file buffer in kilobytes (1–63, default = 32).
/C	Creates a cross-reference (.CRF) file.
/D	Adds a first-pass program listing to *list_file* if a list file was specified (default = second-pass listing only).
/D*symbol*	Defines *symbol* as a null text string.
/E	Assembles code for an 8087/80287 emulator.
/I*path*	Defines a directory to be searched for *include* files.
/L	Creates a list (.LST) file with line-number information.
/ML	Preserves case sensitivity in all symbol names.

(more)

/MU	Converts all lowercase names to uppercase names.
/MX	Preserves lowercase in public and external names only.
/N	Suppresses generation of tables of macros, structures, records, groups, segments, and symbols at the end of the list file.
/P	Checks for impure code in 80286 protected mode; has no effect unless the .286P directive is included in the source file.
/R	Assembles code for an 8087/80287 math coprocessor.
/S	Arranges program segments in order of occurrence.
/T	Selects terse mode, suppressing all messages generated during assembly except error messages.
/V	Selects verbose mode, displaying the number of lines and symbols at the end of assembly.
/X	Includes false conditionals in the list file.
/Z	Displays source lines with errors during assembly.

Note: Switches can be either uppercase or lowercase and can be preceded by a dash (-) instead of a forward slash (/).

Description

MASM translates assembly-language source code into relocatable object modules. The object modules can then be placed in a library file or processed by the Microsoft Object Linker (LINK) to create an executable program.

The *source_file* parameter is the only required filename. It specifies a file containing the assembly-language source code in ASCII text. If no extension is specified, MASM uses .ASM. If no source file is entered in the command line, MASM prompts for a source file name.

The *object_file* parameter specifies the file that will contain the assembled relocatable object code. If this parameter is not supplied, MASM uses the same filename as *source_file* but substitutes the extension .OBJ.

The *list_file* parameter specifies a destination file or device for the optional program listing. The listing contains the original source code, the assembled machine code, macro definitions and expansions, and other useful information, formatted into pages with titles, dates, and page numbers. If the destination of the listing is a file, the file's default extension is .LST. If the *list_file* parameter is not included in the command line, MASM sends the listing to NUL (that is, a listing is not produced).

The *cref_file* parameter specifies the name of a cross-reference file to receive information to be processed by the CREF utility. If a file extension is not specified, MASM uses .CRF. If the *cref_file* parameter is not included in the command line, MASM sends the file to NUL (that is, no cross-reference file is generated).

If the MASM command is entered without parameters, MASM prompts the user for each filename. The default response for each prompt (except the source file prompt) is displayed in square brackets and can be selected by pressing the Enter key.

After the source file is specified, if MASM encounters a semicolon character (;) in the command line or at any prompt, it uses default values for the remaining parameters. MASM ignores any parameters specified after the semicolon.

MASM does two passes to translate the assembly-language code in the source file into relocatable object code. Any errors detected during translation are displayed on standard output and included in the program listing (if one is requested). Two types of errors may be detected: warning errors and severe errors. If MASM encounters a warning error, it still creates the object file, although the resulting file may be unusable. If MASM encounters a severe error, it does not create the object file. After a file has been successfully assembled without errors, the LINK utility can be used to convert the resulting object file into an executable program file.

MASM supports a wide variety of options that can be selected by including switches in the command line or by responding to any prompt.

The /A and /S switches determine the order of segments in the resulting object module file. The /A switch places the segments into the object file in alphabetic order. The /S switch (the default) arranges the segments in the same order they occur in the source file.

The /B*n*, /D*symbol*, and /I*path* switches have rather general effects on the behavior of MASM. The /B*n* switch sets the size (in kilobytes) of the source file's RAM buffer; the value of *n* must be between 1 and 63, inclusive (default = 32). If the RAM buffer is large enough, the entire source file can be kept resident in memory, reducing disk activity during passes. The /D*symbol* switch defines a null text-string symbol from the command line. This symbol can be referenced inside the program with the IFDEF directive to control the conditional assembly of portions of the program. The /I*path* switch specifies a directory that will be searched for files named in assembler INCLUDE statements if those statements do not include an explicit directory. As many as 10 such search paths can be specified with individual /I*path* switches.

The /E and /R switches affect the generation of code for the 8087/80287 emulator or 8087/80287 math coprocessor. (Support for the 80287 is included with MASM versions 3.0 and later.) The /E switch generates software interrupts to floating-point-processor emulator routines. A subprogram assembled with the /E switch can be linked to C, Pascal, and FORTRAN programs and can use the emulator libraries. The /R switch produces in-line machine instructions for the math coprocessor when floating-point mnemonics are used.

The /ML, /MU, and /MX switches control MASM's handling of uppercase and lowercase names. The /ML switch makes MASM case sensitive; that is, it makes MASM differentiate a

name in uppercase letters from the same name in lowercase letters. (The /ML switch should not be used if the source file contains 8087 WAIT instructions and MASM 4.0 is being used to translate the file.) The /MU switch (the default) makes MASM case insensitive; all lowercase letters are converted to uppercase for purposes of assembly. The /MX switch makes MASM case sensitive for public and external names only (names defined with PUBLIC or EXTRN directives). The /MX switch is often used to process assembly-language functions for C programs.

The /P switch checks for impure code segments that will cause problems if the assembled program is run in 80286 protected mode. The switch checks by flagging any instruction that will change a memory location addressed through the processor's CS register. The /P switch has no effect unless the assembly-language source file includes the .286P directive.

The /C, /D, /L, /N, and /X switches control the contents of the program listing and other optional files that are generated as a result of assembly. The /C switch causes the creation of a cross-reference (.CRF) file and the addition of line numbers to the list (.LST) file (if one exists). The /C switch should be included in the command line if the cross-reference file will be used later with the CREF utility to produce a cross-reference listing. The /D switch includes a listing from the first pass as well as a listing from the second pass in the list file if a list file was specified (default = second-pass listing only). By comparing the two listings, the user can isolate an instruction causing a phase error. (A phase error occurs when MASM makes assumptions about addresses, values, or data types on the first pass that are not valid in the second pass.) The /L switch creates a list file with line-number information and gives it the same name as the source file, with the extension .LST. The /N switch suppresses generation of tables — symbols, segments, groups, structures, records, and macros — at the end of a program listing. The /X switch includes statements inside false conditional statements in the list file, allowing conditionals that do not generate code to be displayed. /X has no effect if the .SFCOND or the .LFCOND directive is used in the source file; if the .TFCOND directive is used, the effects of /X are reversed.

Note: The effects of /X are also reversed in MASM version 1.2. In that version, statements within a false conditional are included in the list file by default, and /X will suppress them.

The /T, /V, and /Z switches affect MASM's display on standard output. The /T (terse) switch suppresses messages to standard output, except for messages indicating warning errors or severe errors. The /V (verbose) switch displays information about the number of source lines and symbols at the end of the assembly, in addition to displaying the normal error and symbol space information. The /Z switch displays the actual source lines producing assembly errors (rather than displaying just the error type and line number).

Note: Versions of MASM earlier than 4.0 always show both the source line and the error message.

Return Codes

0 No errors were found during assembly.
1 An error was detected in one of the command-line parameters.
2 The assembly-language source file could not be opened.
3 The list file could not be created.
4 The object file could not be created.
5 The cross-reference file could not be created.
6 An *include* file could not be opened.
7 At least one severe error was detected during assembly. (MASM deletes the invalid object file.)
8 The assembly was terminated because a memory allocation error occurred.
10 An error occurred in defining a symbol (with the /D*symbol* switch) from the command line.
11 Assembly was interrupted by the user's pressing Ctrl-C or Ctrl-Break.

Examples

To assemble the source file CLEAN.ASM in the current drive and directory and place the resulting relocatable object module in the file CLEAN.OBJ without producing a listing or a cross-reference file, type

```
C>MASM CLEAN;  <Enter>
```

The semicolon after the first parameter causes MASM to use the default values for the rest of the parameters.

To assemble the source file CLEAN.ASM, put the object code in a file named CLEAN.OBJ, create a list file named CLEAN.LST, and place information for later processing by the CREF utility in the cross-reference file CLEAN.CRF, type

```
C>MASM CLEAN,CLEAN,CLEAN,CLEAN  <Enter>
```

or

```
C>MASM CLEAN,,CLEAN,CLEAN  <Enter>
```

To use MASM interactively, enter its name without parameters:

```
C>MASM  <Enter>
```

MASM then prompts for all the necessary information. For example, the interactive session on the next page assembles the file HELLO.ASM into the file HELLO.OBJ, producing no listing or .CRF file.

```
C>MASM  <Enter>
Microsoft (R) Macro Assembler  Version 4.00
Copyright (C) Microsoft Corp 1981, 1983, 1984, 1985.  All rights reserved.

Source filename: [.ASM]: HELLO  <Enter>
Object filename: [HELLO.OBJ]:  <Enter>
Source listing [NUL.LST]:  <Enter>
Cross-reference [NUL.CRF]:  <Enter>

  51004 Bytes symbol space free

    0 Warning Errors
    0 Severe Errors
```

Messages

8087 opcode can't be emulated
An 8087 opcode or the operands used with it produced an instruction the emulator cannot support.

Already defined locally
An attempt was made to define a symbol as EXTRN that had already been defined locally.

Already had ELSE clause
An attempt was made to define an ELSE clause within an existing ELSE clause. (ELSE cannot be nested without nesting IF...ENDIF.)

Already have base register
More than one base register was specified within an operand.

Already have index register
More than one index register was specified within an operand.

Block nesting error
A segment, structure, macro, IRC, IRP, REPT, or nested procedure was not terminated properly.

Byte register is illegal
A byte register was used incorrectly in an instruction.

Can't override ES segment
An attempt was made to override the ES segment in an instruction in which this override is invalid.

Can't reach with segment reg
No ASSUME directive was given to make the variable reachable.

Can't use EVEN on BYTE segment
An EVEN directive was used on a segment declared to be a byte segment.

Circular chain of EQU aliases
An alias EQU ultimately points to itself.

Constant was expected
A constant was expected, but an item was received that does not evaluate to a constant.

CS register illegal usage
The CS register was used incorrectly in one of the instructions.

Data emitted with no segment
Code that is not located within a segment attempted to generate data.

Directive illegal in STRUC
All statements within STRUC blocks must be either comments preceded by a semicolon character (;) or one of the define directives (DB, DW, and so on).

Division by 0 or overflow
An expression was encountered that resulted in either a division by 0 or a number too large to be represented.

DUP is too large for linker
Nesting of DUP operators was such that a record too large for LINK was created.

End of file, no END directive
No END statement was encountered, or a nesting error occurred.

Extra characters on line
Superfluous characters were detected on a line after sufficient information to define an instruction was interpreted.

extra file name ignored
The command line contained more than four filename parameters.

Field cannot be overridden
An attempt was made to give a value to a field that cannot be overridden with a STRUC initialization statement.

Forced error
An error was forced with the .ERR directive.

Forced error - expression equals 0
An error was forced with the .ERRE directive.

Forced error - expression not equal 0
An error was forced with the .ERRNZ directive.

Forced error - pass1
An error was forced with the .ERR1 directive.

Forced error - pass2
An error was forced with the .ERR2 directive.

Forced error - string blank
An error was forced with the .ERRB directive.

Forced error - string not blank
An error was forced with the .ERRNB directive.

Forced error - strings different
An error was forced with the .ERRDIF directive.

Forced error - strings identical
An error was forced with the .ERRIDN directive.

Forced error - symbol defined
An error was forced with the .ERRDEF directive.

Forced error - symbol not defined
An error was forced with the .ERRNDEF directive.

Forward reference is illegal
An item was referenced in the operand of an EQU or equal-sign (=) directive before it was defined.

Illegal register value
A specified register value does not fit into the *reg* field (that is, the value is greater than 7).

Illegal size for item
The size of the referenced item is invalid. This error also frequently occurs when an attempt is made to assemble source code written for assemblers with less strict type-checking than that of the Microsoft Macro Assembler (such as early versions of the IBM assembler). The problem can usually be solved by overriding the type of the operand with the PTR operator.

Illegal use of external
A variable that was declared external was used incorrectly.

Illegal use of register
An attempt was made to use a register with an instruction in which a register cannot be used.

Illegal value for DUP count
The DUP count was not a constant that evaluates to a positive integer greater than zero.

Improper operand type
An operand was used in a way that prevents opcode generation.

Improper use of segment register
An attempt was made to use a segment register in an instruction in which use of a segment register is not permitted.

Impure memory reference
An attempt was made to store data in the code segment when the .286P directive and the /P switch were in effect.

Index displ. must be constant
An index displacement was used incorrectly or did not evaluate to an absolute number or memory address.

Internal error
An internal logic error was detected in the assembler. Document the circumstances and contact Microsoft Corporation.

Label can't have seg. override
A segment override was used incorrectly.

Left operand must have segment
The content of the right operand requires that a segment be specified in the left operand.

Line too long expanding *symbol*
A symbol defined by an EQU or equal-sign (=) directive is so long that expanding it will cause the assembler's internal buffers to overflow. This message may indicate a recursive text macro.

Missing data; zero assumed
An operand is missing from a statement and MASM assumes its value is zero. This is a warning error; the object file is not deleted as it is with severe errors.

More values than defined with
Too many initial values were given when defining a variable using a REC or STRUC type.

Must be associated with code
A data-related item was used where a code-related item was expected.

Must be associated with data
A code-related item was used where a data-related item was expected.

Must be AX or AL
A register other than AX or AL was specified where only these are acceptable.

Must be in segment block
An attempt was made to generate code by instructions that were not contained within a segment.

Must be index or base register
An instruction requires a base or index register, and some other register was specified within square brackets ([]).

Must be record field name
A record field name was expected, but something else was encountered.

Must be record or fieldname
A record name or field name was expected, but something else was encountered.

Must be register
A register was expected as the operand, but something else was encountered.

Must be segment or group
A segment or group was expected, but something else was encountered.

Must be structure field name
A structure field name was expected, but something else was encountered.

Must be symbol type
A BYTE, WORD, DWORD, or similar designation was expected, but something else was encountered.

Must be var, label or constant
A variable, label, or constant was expected, but something else was encountered.

Must have opcode after prefix
A REP, REPE, REPNE, REPZ, or REPNZ instruction was not followed by the mnemonic for a string operation.

Near JMP/CALL to different CS
An attempt was made to do a NEAR jump or call to a location in a code segment defined with a different ASSUME:CS.

No immediate mode
Immediate data was supplied as an operand for an instruction that cannot use immediate data. For example, immediate data cannot be moved directly with a MOV instruction to a segment register; it must first be moved into a general register and then copied to the segment register.

No or unreachable CS
An attempt was made to jump to a label that is unreachable.

Normal type operand expected
A STRUC, BYTE, WORD, or some other invalid operand was encountered when a variable label was expected.

Not in conditional block
An ENDIF or ELSE statement was encountered, and no previous conditional-assembly directive was active.

Not proper align/combine type
The SEGMENT parameters are incorrect. Check the align and combine types to be sure they are valid.

One operand must be const
The addition operator was used incorrectly.

Only initialize list legal
An attempt was made to use a STRUC name without angle brackets (<>).

Operand combination illegal
A two-operand instruction was specified and the combination specified was invalid.

Operand must have segment
A SEG directive was used incorrectly.

Operand must have size
An operand was encountered that needed a specified size, but none had been provided. Often this error can be remedied by using the PTR operator to specify a size type.

Operand not in IP segment
An operand cannot be accessed because it is not in the segment last assigned to CS with an ASSUME directive.

Operand types must match
MASM encountered different kinds or sizes of arguments in a case where they must match.

Operand was expected
MASM expected an operand, but an operator was encountered.

Operands must be same or 1 abs
The subtraction operator was used incorrectly.

Operator was expected
MASM expected an operator, but an operand was encountered.

Out of memory
System memory is insufficient to complete the assembly. If a listing (.LST) or cross-reference (.CRF) file was being generated, retry the assembly, generating only an object file. It may also be necessary to modify the source program to reduce the load on the symbol table (by shortening names or reducing the number of EQU statements or macros, for example).

Override is of wrong type
An attempt was made to use a data item of incorrect size in a STRUC initialization statement.

Override value is wrong length
The override value for a structure field is too large to fit in the field.

Override with DUP is illegal
An attempt was made to use DUP to override in a STRUC initialization statement.

Phase error between passes
The program has ambiguous instruction directives that caused the location of a label in the program to change in value between the first and second passes of MASM. A common cause is a forward reference to a typed data item in the instructions preceding the label that generated the phase error message. Use the /D switch to produce a first-pass listing to aid in resolving phase errors between passes.

Redefinition of symbol
This message is displayed during first pass upon the second declaration of a symbol that has been defined in more than one place.

Reference to mult defined
The instruction references a symbol that has been defined more than once.

Register already defined
An internal error was detected. Note the circumstances of the failure and contact Microsoft Corporation.

Relative jump out of range
A conditional jump references a label that is out of the allowed range of –128 to +127 bytes relative to the current instruction. The problem usually can be corrected by reversing the condition of the jump and using an unconditional jump (JMP) to the out-of-range label.

Segment parameters are changed
The list of parameters encountered for a SEGMENT was not identical to the list specified the first time the segment was used.

Shift count is negative
A shift expression was generated that resulted in a negative shift count.

Should have been group name
A group name was expected, but something else was encountered.

Symbol already different kind
An attempt was made to redefine an already defined symbol.

Symbol has no segment
An attempt was made to use a variable with SEG that has no known segment.

Symbol is already external
An attempt was made to redefine a symbol as local that has already been defined as external.

Symbol is multi-defined
This message is displayed during the second pass upon each declaration of a symbol that has been defined in more than one place.

Symbol is reserved word
An attempt was made to use a reserved MASM word as a symbol.

Symbol not defined
A symbol that had not been defined was used.

Symbol type usage illegal
A PUBLIC symbol was used incorrectly.

Syntax error
The syntax of the statement does not match any recognizable syntax.

Type illegal in context
The type specified is of an unacceptable size.

Unable to open input file *filename*
The specified source file cannot be found.

unknown switch *letter*
The command line included an invalid switch.

Unknown symbol type
MASM does not recognize the size type specified in a label or external declaration. Rewrite with a valid type such as BYTE, WORD, or NEAR.

Value is out of range
A value is too large for its expected use.

Wrong type of register
A directive or instruction expected one type of register, but another type was encountered.

DEBUG

Program Debugger

Purpose

Allows the controlled execution of a program for debugging purposes or the alteration of the binary contents of any file. The DEBUG utility is supplied with the MS-DOS distribution disks.

Syntax

DEBUG

or

DEBUG *filename* [*parameter*...]

where:

filename	is the name of the file that contains data to be modified or a program to be debugged. If *filename* includes an extension, it must be specified.
parameter...	is one or more filenames or switches required by a program being debugged.

Description

The DEBUG program allows a file to be loaded, examined, and altered. If the file is not a .EXE file or a .HEX file, it may also be written back to disk. If the file contains a program, the program can be disassembled, modified, traced one instruction at a time, or executed at full speed with preset breakpoints. DEBUG can also be used to read from and write to input/output (I/O) ports and to read, modify, and write absolute disk sectors.

The command line typically includes the *filename* parameter, which is the name of an executable program (with the extension .COM or .EXE) to be loaded into DEBUG's memory buffer. Files with the extension .EXE are loaded in a manner compatible with the MS-DOS loader; if necessary, the contents of the file are relocated so that the program is ready to execute. Files with the extension .HEX are converted to binary images and loaded at the internally specified address. All other files are assumed to be direct memory images and are read directly into memory starting at offset 100H.

An appropriate program segment prefix (PSP) is synthesized at the head of DEBUG's buffer for use by the target program (the program being debugged). The PSP includes a command tail at offset 80H and default file control blocks (FCBs) at offsets 5CH and 6CH, constructed from the optional parameters following *filename*.

After DEBUG is loaded and the first file named in the command line is also located and loaded, DEBUG displays its special prompt character, a hyphen (-), and awaits a command. DEBUG commands consist of a single letter, usually followed by one or more

parameters. Uppercase and lowercase characters are treated the same except when they are contained in strings enclosed within single or double quotation marks. All commands are executed by pressing the Enter key.

The DEBUG commands are

Command	Action
A	Assemble machine instructions (versions 2.0 and later).
C	Compare memory areas.
D	Display memory.
E	Enter data.
F	Fill memory.
G	Go execute program.
H	Perform hexadecimal arithmetic.
I	Input from port.
L	Load file or sectors.
M	Move (copy) data.
N	Name file or command-tail parameters.
O	Output to port.
P	Proceed through loop or subroutine (versions 3.0 and later).
Q	Quit debugger.
R	Display or modify registers.
S	Search memory.
T	Trace program execution.
U	Disassemble (unassemble) program.
W	Write file or sectors.

The parameters for a DEBUG command include addresses, ranges, 8-bit or 16-bit hexadecimal values, and lists. Multiple parameters can be separated by spaces, tabs, or commas, but separators are *required* only between hexadecimal values.

An address can be a simple offset or a complete address in the form *segment:offset*. The offset is always a hexadecimal number in the range 00H through FFFFH; the segment can be either a hexadecimal value in the same range or a two-character segment register name (CS, DS, ES, or SS). If the segment portion of an address is absent, DEBUG uses DS unless an A, G, L, T, U, or W command is used, in which case DEBUG uses CS.

A range specifies an area of memory and can be expressed as either two addresses or a starting address and a length. A segment can be included only in the first element of a range; an error message is displayed if a segment is found in the second address. A length is represented by the letter L, followed by a hexadecimal value between 00H and FFFFH that indicates the number of bytes following the starting address that the command should operate on.

Note: Any length that causes an address to exceed 16 bits will generate an error.

A byte, or 8-bit, value is entered as one or two hexadecimal digits, whereas a word, or 16-bit, value is entered as one to four hexadecimal digits. Leading zeros can be omitted.

A list is composed of one or more byte values or strings, separated by spaces, commas, or tabs. A string is one or more ASCII characters enclosed within single or double quotation marks. Case is significant within a string. If the same type of quote character that is used to delimit the string occurs inside the string itself, the character must be doubled inside the string in order to be interpreted correctly. For example:

```
"This ""string"" is OK."
```

When used, a list must be the last parameter in the command line.

DEBUG responds to an invalid command by pointing to the approximate location of the error with a caret character (^) and displaying the word *Error*. For example:

```
-D CS:0100,CS:0200  <Enter>
             ^ Error
```

DEBUG maintains a set of virtual CPU registers for a program being debugged. These registers can be examined and modified with DEBUG commands. When a program is first loaded for debugging, the virtual registers are initialized with the following values:

Register	.COM Program	.EXE Program
AX	Valid drive error code	Valid drive error code
BX	Upper half of program size	Upper half of program size
CX	Lower half of program size	Lower half of program size
DX	Zero	Zero
SI	Zero	Zero
DI	Zero	Zero
BP	Zero	Zero
SP	FFFEH or top of available memory minus 2	Size of stack segment
IP	100H	Offset of entry point within target program's code segment
CS	PSP	Base of target program's code segment
DS	PSP	PSP
ES	PSP	PSP
SS	PSP	Base of target program's stack segment

Note: DEBUG checks the first three parameters in the command line. If the second and third parameters are filenames, DEBUG checks any drive specifications with those filenames to verify that they designate valid drives. Register AX contains one of the following codes:

Code	Meaning
0000H	The drives specified with the second and third filenames are both valid, or only one filename was specified in the command line.
00FFH	The drive specified with the second filename is invalid.
FF00H	The drive specified with the third filename is invalid.
FFFFH	The drives specified with the second and third filenames are both invalid.

DEBUG also maintains a set of virtual flags, which may be set or cleared. The flags are

Flag Name	Value If Set (1)	Value If Clear (0)
Overflow	OV (Overflow)	NV (No Overflow)
Direction	DN (Down)	UP (Up)
Interrupt	EI (Enabled)	DI (Disabled)
Sign	NG (Minus)	PL (Plus)
Zero	ZR (Zero)	NZ (Not Zero)
Aux Carry	AC (Aux Carry)	NA (No Aux Carry)
Parity	PE (Even)	PO (Odd)
Carry	CY (Carry)	NC (No Carry)

Before DEBUG transfers control to the target program, it saves the actual CPU registers and then loads them with the current values of the virtual registers. Conversely, when control reverts to DEBUG from the target program, the returned register contents are stored back in the virtual register set for inspection and alteration by the user.

Examples

To load the file SHELL.EXE in the current directory for execution under the control of DEBUG, type

```
C>DEBUG SHELL.EXE  <Enter>
```

To use the DEBUG program to inspect or modify memory or to read, modify, and write absolute disk sectors, simply type

```
C>DEBUG  <Enter>
```

Message

File not found
The filename supplied as the first parameter in the DEBUG command line cannot be found.

DEBUG: A

Assemble Machine Instructions

Purpose

Allows entry of assembler mnemonics and translates them into executable machine code.

Syntax

A [*address*]

where:

address is the starting location for the assembled machine code.

Description

The Assemble Machine Instructions (A) command accepts assembly-language statements, rather than hexadecimal values, for the Intel 8086/8088 microprocessors and the Intel 8087 math coprocessor and then assembles each statement into executable machine code.

The *address* parameter specifies the location where entry of assembly-language mnemonics will begin. If *address* is omitted, DEBUG uses the address following the last instruction generated the last time the A command was used. If the A command has not been used, DEBUG uses the current value of the target program's CS:IP registers.

After an A command is entered, DEBUG prompts for each assembly-language statement by displaying the address, in the form of a segment and an offset, in which the assembled code will be stored. When the Enter key is pressed, the assembly-language statement is translated, and each byte of the resulting machine instruction is stored sequentially in memory (overwriting existing information), beginning at the displayed address. The address following the last byte of the machine instruction is then displayed so that the user can enter the next assembly-language statement. Pressing the Enter key alone in response to the address prompt terminates the A command.

The syntax of assembly-language statements accepted by the DEBUG A command differs slightly from that of the usual Microsoft Macro Assembler programming statements. The differences can be summarized as follows:

- All numbers are assumed to be hexadecimal integers and should be entered without a trailing H character.
- Segment overrides must be specified by preceding the entire instruction with CS:, DS:, ES:, or SS:.
- File control directives (NAME, PAGE, TITLE, and so forth), macro definitions, record structures, and conditional assembly directives are not supported by DEBUG.
- Specific hexadecimal values, rather than program labels, must be included.

- When the data type (word or byte) is not implicit in the instruction, the type must be specified by preceding the operand with BYTE PTR (or BY) or WORD PTR (or WO).
- The size of the string in a string operation must be specified by adding a B (byte) or W (word) to the string instruction mnemonic (for example, LODSB or LODSW).
- The DB and DW instructions accept a parameter of the type *list* and assemble byte and word values directly.
- The WAIT or FWAIT opcodes for 8087 assembler statements are not generated by default, so they must be coded explicitly.
- Memory locations are differentiated from immediate operands by enclosing memory addresses in square brackets.
- Repeat prefixes, such as REP, REPZ, or REPNZ, can be entered either alone on the line preceding the statement they affect or immediately preceding the statement on the same line.
- Although the assembler generates the optimal form (SHORT, NEAR, or FAR) for jumps or calls, depending on the destination address, these designations can be overridden by preceding the operand with a NEAR (or NE) or FAR (no abbreviation) prefix.
- The mnemonic for a FAR RETURN is RETF.

Examples

To begin assembling code at address CS:0100H, type

```
-A 100  <Enter>
```

To assemble the instruction sequence

```
LODS  WORD PTR [SI]
XCHG  BX,AX
JMP   [BX]
```

beginning at address CS:0100H, the following dialogue would take place:

```
-A 100  <Enter>
1983:0100  LODSW  <Enter>
1983:0101  XCHG BX,AX  <Enter>
1983:0103  JMP [BX]  <Enter>
1983:0105  <Enter>
```

To continue assembling at the location following the last instruction generated by a previous A command, type

```
-A  <Enter>
```

DEBUG: C

Compare Memory Areas

Purpose

Compares two areas of memory and reports any differences.

Syntax

C *range address*

where:

range	is the starting and ending addresses or the starting address and length of the first area of memory to be compared.
address	is the starting address of the second area of memory to be compared.

Description

The Compare Memory Areas (C) command compares the contents of two areas of memory. The location and contents of any differing bytes are displayed in the following format:

address1 byte1 byte2 address2

If no differences are found, the DEBUG prompt returns.

The *range* parameter specifies the starting and ending addresses or the starting address and length in bytes of the first area of memory to be compared. The *address* parameter specifies the beginning address of the second area of memory to be compared. If a segment is not included in *range* or *address*, DEBUG uses DS.

Example

To compare the 64 bytes beginning at CS:CE00H with the 64 bytes beginning at CS:CF0AH, type

```
-C CS:CE00 CE3F CS:CF0A  <Enter>
```

or

```
-C CS:CE00 L40 CS:CF0A  <Enter>
```

If any differences are found, DEBUG displays them in the following format:

```
2124:CE06  00  FF  2124:CF10
```

DEBUG: D

Display Memory

Purpose

Displays the contents of an area of memory in hexadecimal and ASCII format.

Syntax

D [*range*]

where:

range is the starting and ending addresses or the starting address and length of the area to be displayed.

Description

The Display Memory (D), or Dump, command displays the contents of a specified range of memory addresses in hexadecimal and ASCII format.

The *range* parameter gives the starting and ending addresses or the starting address and length in bytes of the memory to be displayed. If *range* does not include a segment, DEBUG uses DS.

If *range* is omitted the first time the D command is used, the display starts at the target program's CS:IP registers. If *range* was specified in a preceding D command, the memory address following the last address displayed by that command is used. If a length is not explicitly stated in a D command, 128 bytes are displayed.

Each line displays a segment and offset, followed by the contents of 16 bytes of memory represented as hexadecimal values and separated by spaces (except the eighth and ninth values, which are separated by a dash), followed by the ASCII character equivalents (if any) of the same 16 bytes. In the ASCII portion, nonprinting characters are displayed as periods.

Examples

To display the contents of the 128 bytes of memory beginning at 7F00:0100H, type

```
-D 7F00:0100  <Enter>
```

The contents of the memory addresses are displayed in the following format:

```
7F00:0100  20 64 65 76 69 63 65 0D-0A 00 60 39 0D 0A 00 7C    device...'9...|
7F00:0110  39 08 20 08 00 81 39 04-1B 5B 32 4A 42 BD 11 44   9. ...9..[2JB=.D
7F00:0120  2E 26 45 AF 11 47 B3 11-48 A5 11 4C B8 11 4E D3   .&E/.G3.H%.L8.NS
7F00:0130  11 50 DF 11 51 AB 11 54-DF 1E 56 37 11 5F 9F 16   .P_.Q+.T_.V7._..
7F00:0140  24 C0 11 00 03 4E 4F 54-C1 07 0A 45 52 52 4F 52   $@...NOTA..ERROR
7F00:0150  4C 45 56 45 4C 85 08 05-45 58 49 53 54 18 08 00   LEVEL...EXIST...
7F00:0160  03 44 49 52 03 91 0C 06-52 45 4E 41 4D 45 01 C0   .DIR....RENAME.@
7F00:0170  0F 03 52 45 4E 01 C0 0F-05 45 52 41 53 45 01 68   ..REN.@..ERASE.h
```

To view the next 128 bytes of memory, type

```
-D  <Enter>
```

In this case, the contents of memory addresses 7F00:0180H through 7F00:01FFH are displayed.

DEBUG: E

Enter Data

Purpose

Enters data into memory.

Syntax

E *address* [*list*]

where:

address	is the first memory location for data entry.
list	specifies the data to be entered into successive bytes of memory, starting at *address*.

Description

The Enter Data (E) command allows data to be entered into successive memory locations. The data can be entered in either hexadecimal or ASCII format. Data previously stored in the specified locations is lost.

The *address* parameter specifies the first byte to be modified. If *address* does not include a segment, DEBUG uses DS. The address is incremented for each byte of data stored.

The *list* parameter is one or more hexadecimal byte values and/or strings, separated by spaces, commas, or tab characters. Strings must be enclosed within single or double quotation marks, and case is significant within a string.

If *list* is included in the command line, the changes to memory are made unless an error is detected in the command line, in which case an error message is displayed and the E command is terminated. If *list* is omitted from the command line, the user is prompted byte by byte for data to be entered into memory, starting at *address*. The current contents of a byte are displayed, followed by a period. A new value for that byte can be entered as one or two hexadecimal digits (extra characters are ignored) or the contents can be left unchanged. Pressing the spacebar displays the contents of the next byte. Entering a minus sign or hyphen character (-) instead of pressing the spacebar displays the contents of the previous byte. A maximum of 8 bytes can be entered on each input line; a new line is begun each time an 8-byte boundary is crossed. Pressing the Enter key without pressing the spacebar or entering any data terminates data entry.

Text strings can be entered only by using the *list* parameter; they cannot be entered in response to an address prompt.

Examples

To store the byte values 00H, 0DH, and 0AH in the three bytes beginning at DS:1FB3H, type

```
-E 1FB3 00 0D 0A  <Enter>
```

To store the string *MAIN MENU* into memory beginning at address ES:0C14H, type

```
-E ES:C14 "MAIN MENU"  <Enter>
```

DEBUG: F

Fill Memory

Purpose

Stores a repetitive data pattern in an area of memory.

Syntax

F *range list*

where:

range	is the starting and ending addresses or starting address and length of the memory to be filled.
list	is the data to be entered.

Description

The Fill Memory (F) command fills an area of memory with the data from a list. The data can be entered in either hexadecimal or ASCII format. Any data previously stored at the specified locations is lost. If an error message is displayed, the original values in memory remain unchanged.

The *range* parameter specifies the starting and ending addresses or the starting address and hexadecimal length in bytes of the area of memory to be filled. If *range* does not specify a segment, DEBUG uses DS.

The *list* parameter specifies one or more hexadecimal byte values and/or strings, separated by spaces, commas, or tab characters. Strings must be enclosed in single or double quotation marks, and case is significant within a string.

If the area to be filled is larger than the data list, the list is repeated as often as necessary to fill the area. If the data list is longer than the area of memory to be filled, it is truncated to fit into the area.

Examples

To fill the area of memory from DS:0B10H through DS:0B4FH with the value 0E8H, type

```
-F B10 B4F E8  <Enter>
```

or

```
-F B10 L40 E8  <Enter>
```

To fill the 16 bytes of memory beginning at address CS:1FA0H by replicating the 2-byte sequence 0DH 0AH, type

```
-F CS:1FA0 1FAF 0D 0A  <Enter>
```

or

```
-F CS:1FA0 L10 0D 0A  <Enter>
```

To fill the area of memory from ES:0B00H through ES:0BFFH by replicating the text string *BUFFER*, type

```
-F ES:B00 BFF "BUFFER"  <Enter>
```

or

```
-F ES:B00 L100 "BUFFER"  <Enter>
```

DEBUG: G

Go

Purpose

Transfers control from DEBUG to the program being debugged.

Syntax

G [=*address*] [*break0* [... *break9*]]

where:

address	is the location DEBUG begins execution.
break0... *break9*	specify from 1 to 10 temporary breakpoints.

Description

The Go (G) command transfers control from DEBUG to the program being debugged. If no breakpoints are set, the program executes until it crashes or finishes, in which latter case the message *Program terminated normally* is displayed and control returns to DEBUG. (After this message is displayed, the program may need to be reloaded before it can be executed again.)

The *address* parameter can specify any location in memory. If no segment is specified, DEBUG uses the target program's CS register. If *address* is omitted, DEBUG transfers to the current address in the target program's CS:IP registers. An equal sign (=) must precede *address* to distinguish it from the breakpoints *break0*... *break9*.

The parameters *break0*... *break9* are addresses that represent from 1 to 10 temporary breakpoints that can be set as part of the G command. A breakpoint is an address at which execution stops. Breakpoints can be placed in any order, because execution stops at the first breakpoint address encountered, regardless of the position of that breakpoint in the list. Each breakpoint address must contain the first byte of an 8086 opcode. DEBUG installs breakpoints by replacing the first byte of the machine instruction at each breakpoint address with an INT 03H instruction (opcode 0CCH). If the program encounters a breakpoint, execution is suspended and control returns to DEBUG. DEBUG then restores the original machine code to the breakpoint addresses; displays the contents of the registers, the status of the flags, and the instruction pointed to by CS:IP; and displays the DEBUG prompt. If the program executes to completion without encountering any of the breakpoints or stops for any reason other than because it encountered a breakpoint, DEBUG does not replace the INT 03H instructions with the original machine code, and the Load File or Sectors (L) command must be used to reload the original program.

The G command requires that the target program's SS:SP registers point to a valid stack that has at least 6 bytes of stack space available. When the G command is executed, it

pushes the target program's flags and CS and IP registers onto the stack and then transfers control to the target program with an IRET instruction. Thus, if the target program's stack is not valid or is too small, the system may crash.

Examples

To begin execution of the program in DEBUG's buffer at location CS:110AH and set breakpoints at CS:12FCH and CS:1303H, type

```
-G =110A 12FC 1303  <Enter>
```

To resume execution of the program after a breakpoint has been encountered and control has been returned to DEBUG, type

```
-G  <Enter>
```

Messages

bp Error

More than 10 breakpoints were specified in a G command. The command must be entered again with 10 or fewer breakpoints.

Program terminated normally

No breakpoints were encountered and the target program executed to completion. If breakpoints were set, the original program should be restored with the L command.

DEBUG: H

Perform Hexadecimal Arithmetic

Purpose

Displays the sum and difference of two hexadecimal numbers.

Syntax

H *value1 value2*

where:

value1 and *value2* are any two hexadecimal numbers from 0 through FFFFH.

Description

The Perform Hexadecimal Arithmetic (H) command displays the sum and the difference of two 16-bit hexadecimal numbers — that is, the result of the operations *value1+value2* and *value1–value2*. If *value2* is greater than *value1*, the difference of the two values is displayed as a two's complement number. This command is convenient for quickly calculating addresses and other values during an interactive debugging session.

Examples

To display the sum and the difference of the values 4B03H and 104H, type

```
-H 4B03 104  <Enter>
```

This produces the following display:

```
4C07  49FF
```

If the addition produces an overflow, the four least significant digits are displayed. For example, the command line

```
-H FFFF 2  <Enter>
```

produces the following display:

```
0001 FFFD
```

If the second number is bigger than the first, the difference is displayed in two's complement form. For example, the command line

```
-H 1 2  <Enter>
```

produces the following display:

```
0003 FFFF
```

DEBUG: I

Input from Port

Purpose

Reads and displays 1 byte from an input/output (I/O) port.

Syntax

I *port*

where:

port is an I/O port address from 0 through FFFFH.

Description

The Input from Port (I) command reads the specified I/O port address and displays the data as a two-digit hexadecimal number.

Warning: The I command should be used with caution because it directly accesses the computer hardware and no error checking is performed. Input operations directed to the ports assigned to some peripheral device controllers may interfere with the proper operation of the system. If no device has been assigned to the specified I/O port or if the port is write-only, the value displayed by an I command is unreliable.

Example

To read and display the contents of I/O port 10AH, type

```
-I 10A  <Enter>
```

An example of the output of this command is

```
FF
```

DEBUG: L

Load File or Sectors

Purpose

Loads a file or individual sectors from a disk into DEBUG's memory.

Syntax

L [*address*]

or

L *address drive start number*

where:

address	is the memory location for the data to be read from the disk.
drive	is the number of the disk drive to read (0 = drive A, 1 = drive B, 2 = drive C, and so on).
start	is the hexadecimal number of the first logical sector to load (0–FFFFH).
number	is the hexadecimal number of consecutive sectors to load (0–FFFFH).

Description

The Load File or Sectors (L) command loads a file or individual sectors from a disk. When the L command is entered without parameters or with only an address, the file specified in the DEBUG command line or the one in the most recent Name File or Command-Tail Parameters (N) command line is loaded from the disk into memory. If no segment is specified in *address*, DEBUG uses CS. If the file's extension is .EXE, the file is placed in DEBUG's target program buffer at the load address specified in the .EXE file's header. If the file's extension is .COM, the file is loaded at offset 100H. (If for some reason an address other than 100H is entered for a .EXE or .COM file, an error message is displayed; if the address is 100H, the specification is ignored.) The length of the file or, in the case of a .EXE file, the actual length of the program (the length of the file minus the header) is placed in the target program's BX and CX registers, with the most significant 16 bits in register BX.

The L command can also be used to bypass the MS-DOS file system and directly access logical sectors on the disk. The memory address (*address*), disk drive number (*drive*), starting logical sector number (*start*), and number of sectors to load (*number*) must all be specified in the command line.

Note: The L command should not be used to access logical sectors on network drives.

Examples

To load the file specified in the DEBUG command line or in the most recent N command into DEBUG's target program buffer, type

```
-L  <Enter>
```

To load eight sectors from drive B, starting at logical sector 0, to memory location CS:0100H, type

```
-L 100 1 0 8  <Enter>
```

Messages

Disk error reading drive *X*
The specified drive does not exist or the disk in the specified drive is defective.

File not found
The file specified in the most recent N command cannot be found.

DEBUG: M

Move (Copy) Data

Purpose

Copies the contents of one area of memory to another.

Syntax

M *range address*

where:

range	specifies the starting and ending addresses or the starting address and length of the area of memory to be copied.
address	is the first byte in which the copied data will be placed.

Description

The Move (Copy) Data (M) command copies data from one memory location to another without altering the data in the original location. If the source and destination areas overlap, the data is copied so that the resulting copy is correct; the data in the *original* location is changed where the two areas overlap.

The *range* parameter specifies either the starting and ending addresses or the starting address and length of the memory to be copied. The *address* parameter is the first byte in which the copy will be placed. If *range* does not contain an explicit segment, DEBUG uses DS; if *address* does not contain a segment, DEBUG uses the segment used for *range*.

Example

To copy the data in locations DS:0800H through DS:08FFH to locations DS:0900H through DS:09FFH, type

```
-M 800 8FF 900  <Enter>
```

or

```
-M 800 L100 900  <Enter>
```

DEBUG: N

Name File or Command-Tail Parameters

Purpose

Inserts filenames and/or switches into the simulated program segment prefix (PSP).

Syntax

N *parameter* [*parameter*...]

where:

parameter is one or more filenames or switches to be placed in the simulated PSP.

Description

The Name File or Command-Tail Parameters (N) command is used to enter one or more parameters into the simulated PSP that is built at the base of the buffer holding the program to be debugged. The N command can also be used before the Load File or Sectors (L) and Write File or Sectors (W) commands to name the file to be read from or written to a disk.

The count of the characters following the N command is placed at DS:0080H in the simulated PSP, and the characters themselves are copied into the PSP starting at offset 81H. The string is terminated by a carriage return (0DH), which is not included in the count. If the first and second parameters follow the naming conventions for MS-DOS files, they are parsed into the default file control blocks (FCBs) in the simulated PSP at offsets 5CH and 6CH, respectively. (Switches specified as parameters are stored in the PSP starting at offset 81H along with the rest of the command line but are not included in the FCBs.)

If the N command line contains only one filename, any parameters placed in the default FCBs by a previous N command are destroyed. If the drive specified with the first filename parameter is invalid, the AL register is set to 0FFH. If the drive specified with the second filename parameter is invalid, the AH register is set to 0FFH. The existence of a file specified with the N command is not verified until it is loaded with the L command.

Examples

Assume that DEBUG was started without specifying the name of a target program in the command line. To load the program CLEAN.COM for execution under the control of DEBUG, use the N and L commands together as follows:

```
-N CLEAN.COM  <Enter>
-L  <Enter>
```

Then, to place the parameter MYFILE.DAT in the simulated PSP's command tail and parse MYFILE.DAT into the first default FCB, type

```
-N MYFILE.DAT  <Enter>
```