Microsoft® Windows

Software Development Kit

Programmer's Utility Guide

Version 1.03

Microsoft Corporation

Document Number 050051051-103-I01-1086

Contents

Figures

Tables

Chapter 1
Introduction

1.1 Introduction

The Microsoft® Windows Software Development Kit is a collection of utilities, debugging aids, and sample source programs that will help you begin development of Windows applications. This guide explains how to install the development kit on your computer and how to use the programming utilities to create applications.

Since some of the programming utilities are Windows applications, you must install Windows on your computer. You also need Windows when you debug your application. This chapter explains how to install Windows.

For information on using Windows, see the *Microsoft Windows User's Guide*. This guide explains such things as how to start Windows, how to use Windows menus and dialog boxes, and how to run applications. It also describes a key element of the Windows user interface, the MS-DOS Executive, as well as how to use the Control Panel and the Spooler to adjust system settings and examine the printing queue. In addition, the guide provides information on using PIF (program information) files to run applications that were not written specifically for Windows.

1.2 Development Kit Contents

The Microsoft Windows Software Development Kit includes the following:

- Utilities Disk 1
- Utilities Disk 2
- Libraries and Include Files Disk (C)
- Libraries, Include Files, and Sample Source Disk (Pascal)
- Symbol and Debug Files
- Sample Source Code Disk 1
- Sample Source Code Disk 2
- *Microsoft Windows Update to Programmer's Reference and Programming Guide*

- *Microsoft Windows Programmer's Utility Guide*
- *Microsoft Windows Quick Reference*
- *Microsoft Windows Programming Guide*
- *Microsoft Windows Application Style Guide*
- *Microsoft Windows Programmer's Reference*

For a complete listing and explanation of the files on the Windows Software Development Kit disks, see Appendix A, "Disk Contents."

1.3 About this Guide

The *Microsoft Windows Programmer's Utility Guide* is divided into two sections: "DOS Utilities" and "Windows Development Applications." The "DOS Utilities" section describes programs that perform such actions as compiling and linking your application's source code and debugging and maintaining application files. Some of these programs can be run from Windows, but unlike the Windows development applications, the DOS utilities use command lines and parameters for input. "DOS Utilities" includes descriptions of the following programming utilities:

Utility	Description
cl	Compiles C-language application source files
pascal	Compiles Pascal application source files
masm	Assembles assembly-language application source files
rc	Compiles application resource files
link4	Links the compiled source files for applications
symdeb	Debugs applications
make	Maintains assembly- and high-level-language programs

Windows development applications are used just like other Windows applications. They have menus with commands that can be chosen, and they have dialog boxes for input. The "Windows Development Applications" section describes the following programming utilities:

Utility	Description
Font Editor	Creates fonts for applications
Icon Editor	Creates cursors, icons, and bitmaps for applications
Dialog Editor	Creates dialog boxes for applications
Shaker Program	Shows the effect of memory movement on applications
Heapwalker Program	Opens the global heap for examination

1.4 What You Need

To use the development kit on your system, you must have the following:

- A personal computer that supports Microsoft Windows
- 512K memory
- A floppy disk drive configured as drive A
- A hard disk drive configured as drive C, or second floppy disk drive configured as drive B
- A graphics monitor (monochrome or color) and a graphics adapter card. The graphics adapter card you use depends on the machine you are using to run Windows. Examples of cards you might use are an IBM Color Graphics Adapter, a Hercules Graphics Adapter, or an IBM Enhanced Graphics Adapter.

A mouse is recommended but not required. However, a mouse is required to use the Dialog Editor, the Font Editor, or the Icon Editor. (The word "mouse" is used in this guide to refer to a mouse or any similar pointing device.)

For debugging with **symdeb**, an external console or an additional monochrome monitor with display adapter is required.

You must have the following software:

- DOS 2.x or 3.x
- Microsoft Windows, version 1.03 or later
- At least one of the following:

  Microsoft C Compiler, version 4.0

  Microsoft Pascal Compiler, version 3.3 or later

  Microsoft Macro Assembler, version 4.0

Before installing the development kit, make sure that all required hardware and software have been installed. For instructions on how to install the required hardware, use the hardware guide or manual provided by the manufacturer with each device.

Important

If you have a Microsoft Bus Mouse and Windows does not respond to mouse movements, you may need to change the jumper on the mouse's printed circuit board. Generally, you should set the jumper to IRQ2 for the IBM PC XT and to IRQ5 for the IBM PC AT. See the manual that accompanies your mouse bus card for details on how to change the jumper.

1.5 Backing Up Your Disks

Before you begin to install the development kit, you should make working copies of your Windows Software Development Kit disks, using the **copy** or **diskcopy** utility supplied with DOS. Save the original disks for backup.

1.6 Installing the Development Kit

To install the development kit, you need to perform the following tasks:

- Create development directories on your hard disk
- Copy the files on the development kit disks to the development directories on your hard disk
- Create or modify the *config.sys* file to define the proper operating-system configuration
- Create or modify the *autoexec.bat* file to define the proper operating environment

Sections 1.6.1–1.6.5 explain the recommended procedure for installation.

1.6.1 Creating the Development Directories

You create the development directories on your hard disk by using the DOS **mkdir** command. The following is a list of the recommended directories and their contents:

Directory	Contents
c:\bin	All DOS executable files
c:\include	All include (*.inc* and *.h*) files
c:\lib	All library (*.lib*) files
c:\temp	All temporary files created during development
c:\windows	All Windows run-time files and development subdirectories
c:\windows\c	All source files for the sample C applications
c:\windows\cardfile	All source files for the Cardfile application
c:\windows\pascal	All source files for the sample Pascal application

1.6.2 Copying Files

You copy the development kit files from Windows Software Development Kit disks to the development directories by using the DOS **copy** command. Use the following lists to copy files to their recommended directories:

c:\bin

exehdr.exe
implib.exe
lib.exe
link4.exe
make.exe
mapsym.exe
patchdbg.exe
patchrtl.exe
rc.exe
rcpp.exe
symdeb.exe
winstub.exe

c:\include

cmacros.inc
style.h
windows.h
windows.inc
winnames.inc

c:\lib

clibw.lib
cwinlibc.lib
llibw.lib
lwinlibc.lib
mlibw.lib
mwinlibc.lib
pascal.lib
paslibw.lib
slibw.lib
swinlibc.lib

c:\windows

atrm1111.fnt
dialog.exe
fontedit.exe
gdi.sym
heapwalk.exe
iconedit.exe
kernel.sym
shaker.exe
user.sym

c:\windows\c

clock
clock.c
clock.def
clock.h
clock.ico
clock.rc
clockdat.asm
comm.c
declare.h
dlgopen.c
dlgsave.c
fonttest
fonttest.c
fonttest.def
fonttest.h
fonttest.ico
fonttest.rc
gettime.asm
hello
hello.c
hello.def
hello.h
hello.ico
hello.rc
mapmodes
mapmodes.c
mapmodes.def
mapmodes.h
mapmodes.ico
mapmodes.rc
motion
motion.c
motion.def
motion.h
motion.ico
motion.lnk
motion.rc
motion1.c
motion2.c
motioni.c
motlib1.c
motlib1.def
motlib2.c
motlib2.def
print.c
sample
sample.c
sample.def
sample.h
sample.ico
sample.lnk
sample.rc
shapes
shapes.c
shapes.def
shapes.doc
shapes.h
shapes.ico
shapes.rc
tempinit.c
template
template.def
template.h
template.ico
template.rc
tempnres.c
tempres.c
tools.asm
track
track.c
track.def
track.doc
track.h
track.ico
track.rc
type
type.c
type.def
type.doc
type.h
type.ico
type.rc
windwp.c

c:\windows\cardfile

asmsubs.asm
cardfile
cardfile.def
cardfile.h
cardfile.ico
cardfile.lnk
cardfile.rc
cfbitmap.c
cfcard.c
cfclip.c
cfdata.c
cfdb.c
cfdial.c
cfdos.asm
cffile.c
cffind.c
cfinput.c
cfmain.c
cfnew.c
cfopen.c
cfpaint.c
cfprint.c
cfres.c
cfscroll.c
cftext.c
declare.h
dlgopen.c

c:\windows\pascal

muzzle
muzzle.cur
muzzle.def
muzzle.ico
muzzle.inc
muzzle.pas
muzzle.rc

1.6.3 Setting Up the Config.sys File

You set up the operating-system configuration for the development kit by creating or modifying the *config.sys* file. The file should include the following lines:

```
FILES=20
BUFFERS=40
```

The FILES line lets the system have up to 20 open files at a time.

1.6.4 Setting Up the Environment

You set up the operating environment by assigning values to the LIB, INCLUDE, TMP, TEMP, and PATH environment variables with the DOS **set** command. The environment variables are used by the development utilities to locate the files needed to create and execute Windows applications. The variables have the following meanings:

Variable	Meaning
LIB	Used by the development linker, **link4**, to locate library files
INCLUDE	Used by your compiler to locate include files
TMP	Used by your compiler to locate the temporary file directory
TEMP	Used by Windows to locate the temporary file directory
PATH	Used by DOS to locate executable files

To make sure the environment is set up for Windows development each time you start your system, add the following commands to the *autoexec.bat* file:

```
set LIB=c:\lib
set INCLUDE=c:\include
set TMP=c:\temp
set TEMP=c:\temp
set PATH=c:\bin;c:\windows;c:\lib
```

Important

The LIB variable should also specify the directories that contain your compiler's run-time libraries.

The PATH variable should also specify the directories that contain your compiler's executable files.

1.6.5 Restarting the System

After you have installed the development kit and created or modified the *config.sys* and *autoexec.bat* files, reboot the system so the new configuration and environment will take effect.

1.7 Upgrading Your Development Environment to Version 1.03

If you have been using an earlier version of the Windows Software Development Kit, you will need to upgrade the version you are using. To upgrade, you need to delete various files and replace them with the ones included in this package and in version 4.0 of the Microsoft C Compiler. To ensure that version 1.03 of the Windows Software Development Kit works correctly, be sure to complete all of the steps described below.

1. Delete all files relating to the Microsoft C Compiler, version 3.0. You will need version 4.0 of the Microsoft C Compiler to write C applications with version 1.03. Delete the following files:

cc.exe

p1.exe

p2.exe

p3.exe

Copy the following Microsoft C Compiler files to your computer:

cl.exe

c1.exe

c2.exe

c3.exe

2. Delete all Windows executable (*.exe*) files and replace them with the executables on the enclosed disks.

3. Delete all library (*.lib*) files. Replace these with the library files on the enclosed disks and the library files from version 4.0 of the Microsoft C Compiler. When creating your applications, you can now use the C run-time library files supplied with the Microsoft C Compiler. These could not be used with earlier versions of Windows.

4. Delete all include (*.h* and *.inc*) files, and replace them with the include files on the enclosed disks. If you are using C run-time routines, you should also copy the include files from version 4.0 of the Microsoft C Compiler.

5. Delete the existing debugging version of Windows, and replace it with a new debugging version. Follow the steps listed in Section 1.8.1.

6. Delete all symbol (*.sym*) files and replace them with the symbol files on the enclosed disks.

7. Edit any **make** files you may have created using an earlier version of Windows. These files may contain references to files in version 3.0 of the Microsoft C Compiler or to old library files. Replace these filenames with the new ones described in step 1. For information about using the Microsoft C Compiler, see Chapter 2, "C, Pascal, and the Macro Assembler." For information about library files, see Chapter 4, "Windows Linker: Link4."

1.8 Setting Up Windows

You create an executable version of Windows for your computer by using the **setup** program provided on the Microsoft Windows Setup disk. The Setup disk is one of the disks provided with the Microsoft Windows retail product and is not provided with the Microsoft Windows Software Development Kit.

The **setup** program combines files from the retail disks to make a version of Windows that is tailored to your computer.

The **setup** program lets you make two versions of Windows: the standard Microsoft distribution version and a debugging version for running and debugging your Windows applications.

For development purposes, you should make a debugging version of Windows. A standard version is required only if you want to use Windows without full debugging support. Sections 1.8.1 and 1.8.2 explain how to set up each version.

1.8.1 Setting Up the Debugging Version

The debugging version of Windows provides error checking and diagnostic messages useful for debugging applications. It checks for and warns about invalid handles, illegal access to read-only memory, and other serious errors. It also displays messages to show when Windows loads code or moves code and data. All messages are written to your computer's auxiliary (AUX) communication port and can be viewed by connecting a terminal to this port. (For a list of diagnostic messages, see Appendix B, "Diagnostic Messages.")

To create a debugging version of Windows, follow these steps:

1. Prepare a Setup disk for the debugging version by following these steps:
   a. Use the DOS **copy** or **diskcopy** command to make a copy of the Microsoft Windows Setup disk. Put the original working copy in a safe place.
   b. Copy the file *kernel.exe* from the Windows Software Development Kit disk labeled "Symbol and Debug Files" to the new Setup disk.

c. Use the DOS **copy** or **diskcopy** command to make a copy of the Microsoft Windows Build disk. Put the original working copy in a safe place.

d. Copy the files *user.exe* and *gdi.exe* from the Windows Software Development Kit disk labeled "Symbol and Debug Files" to the new Build disk.

2. Put the new Setup disk in drive A and close the door.
3. Change to drive A.
4. Type the following and press the ENTER key to start the **setup** program:

```
setup
```

The **setup** program displays a series of screens that ask you for information about your computer. When asked what disk type and directory to use, answer with the following information:

```
Disk Type
```

Type *h* for hard disk.

```
Directory
```

Type *windows* or the name of the development directory you created when installing the development kit.

For all other questions, answer as appropriate for your computer.

1.8.2 Setting Up the Standard Version

To create a standard version of Windows, use the **setup** program on the Setup and Build disks included with the Microsoft Windows retail product, and follow the steps described in the *Microsoft Windows User's Guide*. If you have already created a debugging version of Windows, make sure that you specify a different directory for the standard version; otherwise **setup** will overwrite the debugging version.

1.9 Tips for Improving the Windows Development Environment

The following is a list of tips for creating a development environment that gives the best possible performance when creating Windows applications:

- Use a PC with an 80286 CPU and at least an 8 Mhz crystal; for example, an 8 Mhz PC AT or compatible.
- Use fast memory that does not require wait states.
- Use a two-to-one disk interleave with the 8 Mhz crystal.
- Use a local-area network. A network provides a uniform development environment and gives cooperating developers fast access to sources.
- Use many small subdirectories. File open-and-read performance decreases with the number of files in the directory.
- Remove as much text from the *windows.h* file as you can. This will reduce the amount of time spent in the C-compiler preprocessor.
- Use a source-code-control system.

DOS Utilities

Chapter 2
Cc, Pascal, and the Macro Assembler

2.1 Introduction

A complete application for the Microsoft Windows operating environment begins with the application's program and resource source files. You can write Windows applications in C, Pascal, or assembly language. You can create resources for an application, such as icons, cursors, menus, and dialog boxes, by using the Windows resource compiler (**rc**) described in Chapter 3, "Resource Compiler: Rc."

To create a Windows application, you must follow these steps:

1. Use a text editor to create your application's C, Pascal, or assembly-language source files.
2. Compile your source files.
   - Use the Microsoft C Compiler to compile any application source files written in C.
   - Use the Microsoft Pascal Compiler to compile any application source files written in Pascal.
   - Use the Microsoft Macro Assembler to assemble any application source files written in assembly language.
3. Use the development utilities Icon Editor, Font Editor, and Dialog Editor to create resources for your application.
4. Use a text editor to create a resource script file listing (or defining) the resources. The resource script file is described in Chapter 3, "Resource Compiler: Rc."
5. Use the Windows resource compiler (**rc**) to compile your application's resource script file.

Once you have completed these steps, you are ready to create a module definition file for your application and to link it. Linking is described in Chapter 4, "Windows Linker: Link4."

This chapter explains how to create source files for Windows applications and how to compile or assemble them.

2.2 C-Language Applications

C-language Windows applications are ordinary C-language programs that use Windows functions, data types, and programming conventions. You compile C-language Windows programs using the Microsoft C Compiler and the **cl** command. All C-language Windows applications must include the *windows.h* file, which contains definitions for all Windows functions, data types, and constants. To include the file, use the #**include** directive at the beginning of each source file.

Example

```
#include "windows.h"
```

2.2.1 Small-, Medium-, Compact-, and Large-Model Applications

Windows applications can use the small, medium, compact, or large programming model. You choose a programming model by supplying an appropriate option when you compile the application source files. You base your choice on your application's need for data and code. The following list describes each programming model and names the compiler option used to generate that model.

Model	Description
Small	This application has one code segment and one data segment. They typically are small applications that cannot be divided easily into separate code segments. Use the **–AS** option, if desired. The option really is not needed since the compiler generates small-model applications by default.
Medium	This application can have several code segments, but only one data segment. Medium-model applications are large applications that swap code segments to conserve memory. Use the **–AM** option.
Compact	This application can have several data segments, but only one code segment. Compact-model applications typically have a large number of data and

	a small number of program statements. Use the **–AC** option.
Large	This application can have several code and data segments. A typical large-model application is a large C program that uses more than 64 kilobytes of data storage. Use the **–AL** option.

Windows requires that all data segments of compact- and large-model applications be fixed. This means the following statement must be in the module definition file of any compact- or large-model application.

```
DATA FIXED
```

See Chapter 4, "The Windows Linker: Link4," for information about the module definition file.

The sample application Hello provided with the Windows Software Development Kit includes examples of how to build a Windows application for each of the four programming models. The *hello.c* source file contains test code that calls various parts of the standard Microsoft C run-time library.

2.2.2 Pascal Calling Conventions

Windows uses the Pascal calling conventions. Therefore, all functions within an application that can be called by Windows must be defined with the **pascal** keyword. The **pascal** keyword ensures that the C function accesses arguments correctly. Functions that can be called by Windows are the **WinMain** function, the application's window functions, and all call-back functions that an application passes to Windows.

2.2.3 The WinMain Function

All C-language Windows applications must define a **WinMain** function. This function is the entry point, or starting point, for the application. It contains statements and Windows function calls that create windows and read and dispatch input intended for the application. The function definition has the following form:

```
int PASCAL WinMain(hInstance,hPrevInstance,lpCmdLine,nCmdShow)
HANDLE hInstance;
HANDLE hPrevInstance;
LPSTR lpCmdLine;
int nCmdShow;
{
        .
        .
        .
}
```

The **WinMain** function must be declared with the **pascal** keyword. Although Windows calls the function directly, **WinMain** must not be defined with the **far** keyword unless it is in a medium- or large-model application.

2.2.4 Callback Functions

Callback functions are functions in an application that Windows calls to carry out specific tasks. An example is a window function that processes messages for an application's windows.

Windows expects callback functions to use the Pascal calling conventions, so the function must be defined with the **pascal** keyword. Windows also uses far function calls to access callback functions. This means that callback functions in small- and compact-model Windows applications must be defined with the **far** keyword to ensure that the function uses a correct return address. The following example shows the form of a callback function:

```
long FAR PASCAL HelloWndProc(hWnd, message, wParam, lParam)
HWND hWnd;
unsigned message;
WORD wParam;
LONG lParam;
{
        .
        .
        .
}
```

Callback functions require special code at their beginnings and ends. This code, called the Windows prolog and epilog, ensures that the correct data segment is used by the function when it executes. To make sure the Windows prolog and epilog are provided, the **–Gw** option must be used with the **cl** command when compiling any application source files containing functions that can be called by Windows.

All callback functions must be listed in the **EXPORTS** statement of the application's module definition file. This identifies the function as a callback and permits Windows to insert the proper data-segment address in the function's prolog when it loads the application.

Local functions (functions used exclusively by the application and not called by Windows) do not require the Windows prolog and epilog, and can use the ordinary C calling conventions.

2.2.5 Compiling Windows Applications

Windows application source files are compiled using the **cl** command. Since the object files generated by **cl** must be linked with the Windows linker (**link4**) the **–c** option should be used to prevent **cl** from attempting to create a file that is not executable under Windows. Other options, such as those specifying programming model, packed structures, Windows prolog and epilog, should be given when the code to be generated requires these features.

Example

```
cl -c -AS -Gsw -Os -Zdp test.c
```

In this example, the source file *test.c* is compiled using the recommended **cl** options for a small-model Windows application source file.

2.2.6 Segment Names

When compiling medium-, compact-, and large-model source files for Windows applications, you must specify the name of the code and data segments to which the given source belongs. You specify the name by using the **–NT** and **–ND** options. If the options are not used, the C compiler assumes the source belongs to the standard code and data segments, _TEXT and _DATA.

2.2.7 Stack Probes

Unless the **–Gs** option is given, the C compiler inserts a stack probe in each function. A stack probe is code that checks the stack to make sure that it has sufficient space for the local variables declared within the function. If the stack would overflow, the code calls the **FatalExit** function

and terminates Windows. Stack probes can be used in Windows applications and libraries. Since libraries use the stack of the caller, a stack probe in a library function checks the caller's stack.

2.2.8 Optimizing for Size

By default, the C compiler optimizes for program speed as it compiles the application sources. Since Windows is a multitasking environment and the size of an application affects the number of applications that can run at a given time, it is better to optimize for size, therefore the **–Os** option is recommended to direct the compiler to optimize for code size instead of speed. If no optimizing is desired, the **–Od** option should be used.

2.2.9 Source-Level Debugging

Windows applications written in C are easier to debug if line-number information is added to the object file. Line-number information can be used by the symbolic debug utility (**symdeb**) to display program lines from the source file when debugging an application. (For more information on **symdeb**, see Chapter 5, "Symbolic Debugging Utility: Symdeb.") To add line-number information, use the **–Zd** option.

2.2.10 Packed Structures

All Windows functions that use structures use packed structures. A packed structure is any structure in which the extra bytes typically used by the C compiler for padding have been removed. Windows applications that use structures with Windows must use the **–Zp** option to direct the compiler to pack bytes in the structures.

2.2.11 Windows and C-Language Libraries

After your application sources have been compiled, you must link the object files with the appropriate C-language libraries for Windows and C run-time libraries. The C-language libraries for Windows, *slibw.lib*, *mlibw.lib*, *clibw.lib*, and *llibw.lib*, contain code for the Windows application startup routines and references for the Windows functions. The C run-time libraries, *slibc.lib*, *mlibc.lib*, *clibc.lib*, and *llibc.lib*, contain code for routines called by the Windows startup routines and for any C run-time functions used by the application. The C-language libraries you link with

depend on your application's programming model. For example, a small-model application must be linked with the small-model libraries *slibw.lib* and *slibc.lib*. Although you must use the Windows linker, **link4**, to link your application, the C compiler adds default library information to your application's object files, so the only library you need to specify in the **link4** command line is the appropriate C-language library for Windows. For more information about linking, see Chapter 4, "The Windows Linker: Link4."

2.2.12 Environment and Call Arguments

The startup routines for Windows applications use the **_setargv** and **_setenvp** functions to copy your application's command line arguments and the current values of the system's variables to the **__argc**, **__argv**, and **environ** variables. These variables can be used by any application by supplying the following definitions in the application source:

```
int __argc;
char *__argv[ ];
char *environ;
```

The **__argc**, **__argv**, and **environ** variables actually occupy space in your application's data segment. If your application does not refer to these variables, you can prevent **_setargv** and **_setenvp** from allocating this space by defining the following functions in your application:

```
void near _setargv( ) { }
void near _setenvp( ) { }
```

These functions must belong to your application's _TEXT segment (the default code segment if the **–NT** option is not given in the **cl** command line).

After using the **__argc**, **__argv**, and **environ** variables, an application can reclaim the space occupied by the variables by using the following statements:

```
environ = free ( environ );
__argv  = free ( __argv );
__argc  = 0;
```

All Windows applications receive a pointer to the environment allocated for the initial load of *win100.bin*. This means the **__argv**[0] value always points to a string that contains *win100.bin* as the filename of the program being run.

2.2.13 C Run-time Functions

Windows applications can call C run-time functions to carry out tasks such as memory allocation and file input and output. However, since the C run-time library was developed for programs running under DOS not Windows, there are some restrictions.

Windows applications can use any input or output function that does not access the system display or keyboard. Input and output functions such as **fread**, **fwrite**, **fclose**, **fprintf**, **fscanf**, and **fgets** can be used to read from and write to disk files, but should not be used to access the keyboard, system display, or communications ports. Functions that access the standard input and output files, such as **printf** and **gets**, should not be used. Although you can use the C run-time input and output functions, a better solution is to develop assembly-language routines that provide raw block input and output through DOS system calls. No matter how an application accesses disk files, it must not keep disk files open for long periods of time, and must be sure to close a disk file before relinquishing control to another application.

Although Windows applications can call C run-time memory-allocation functions, such as **malloc** and **calloc**, the linker replaces these calls with appropriate calls to Windows memory-management functions, such as **LocalAlloc**. Windows memory-management functions are similar but not identical to the C run-time functions, so care must be taken when using the C run-time functions in your applications.

2.2.14 Floating-Point Support

Windows applications that use floating-point variables (that is, variables having **float** type) must specify the **–FPa** option in the **cl** command line. This option directs the compiler to generate the correct floating-point code for Windows applications. The option also adds the proper floating-point library name to the object file (*slibfa.lib*, *mlibfa.lib*, *clibfa.lib*, or *llibfa.lib*) so that the linker automatically links the application with the correct library. Other floating-point libraries, such as *slibfp.lib*, and the floating-point emulator, *em.lib*, must not be used with Windows applications.

2.2.15 Windows Libraries

Windows libraries written in C have slightly different requirements than do Windows applications written in C. Unlike Windows applications, Windows libraries are not executable programs; that is, although a library is loaded, it does not run. Instead, the code in a library is made available to all applications that need to use it, and an application can execute a portion of the library by calling one of the exported functions in the library. Since a Windows library is not a program, it must not have a **WinMain** function. No entry point is required unless the library must carry out some initial task such as initializing a local heap. If an entry point is required, the library must define its own. Information about the entry point and the parameters passed to it can be found in the *Microsoft Windows Programming Guide*.

Exported functions in Windows libraries must have the same attributes as callback functions in Windows applications. The function must use the **far** keyword. Although the Pascal calling convention is optional, it is strongly recommended in order to remain consistent with the Windows interface. Exported functions must have the Windows prolog and epilog, so the **–Gw** option is required. Exported functions must be listed in the library's module definition file.

Since Windows libraries do not use the same startup routine, they should be linked with the C-language libraries for Windows libraries (*swinlibc.lib*, *mwinlibc.lib*, *cwinlibc.lib*, and *lwinlibc.lib*) instead of the libraries for Windows applications. These libraries contain references to the Windows kernel functions and to the C run-time functions only. It is assumed that a Windows library does not require access to Windows user or GDI functions. If a library does require access to those functions, it can be linked with *slibw.lib*, *mlibw.lib*, *clibw.lib*, or *llibw.lib*, as appropriate; however, this library must be specified after the corresponding *winlibc.lib* file on the **link4** command line.

Windows libraries always use the stack of the calling application for parameters and local variables. This means that the value of the **ds** and **ss** registers are not equal when the library is executed. Since the C compiler generates code that assumes the **ds** and **ss** registers are equal, Windows libraries may fail unless compiled with the **–Aw** option. This option directs the compiler to generate code that does not assume that the registers are equal. The following example shows the recommended options for compiling Windows libraries:

```
cl -c -Aw -As -An -Os -Zdp testlib.c
```

In this example, the **–As** and **–An** options are used to complete the programming model specification, since the **–AS** option cannot be used with the **–Aw** option.

Code sharing also restricts which C run-time functions a library can use. Windows libraries must not call C run-time functions that assume that **ds** and **ss** are equal. Appendix C, "C Run-time Functions," contains a list of functions that indicates which functions can and cannot be used by Windows libraries.

Since Windows libraries use the stack of the caller and cannot determine the size of that stack, functions should avoid declaring exceptionally large local variables. Large variables should be declared as static variables within the library's own data segment.

To avoid problems in Windows libraries that use pointers, either by generating their own or receiving them from applications, you should use the following guidelines:

- Always cast pointers to full 32-bit segment:offset addresses. The *windows.h* file contains a variety of type definitions that can be used to cast pointers.
- Use the **–Aw** option when you compile to ensure that pointers receive their proper segment address when cast to 32-bit addresses.
- Make sure functions that receive pointers from an application or from other functions within the library receive long pointers.

2.3 Pascal-Language Applications

Pascal-language Windows applications are ordinary Pascal-language programs that use Windows functions, data types, and programming conventions. You compile Pascal-language Windows programs using the Microsoft Pascal compiler.

2.3.1 Windows Interface

All Pascal Windows applications must include the Windows interface in their source files. The Windows interface contains definitions for the Windows functions, data types, and constants. You can include the Windows interface by using the **INTERFACE** statement and the *windows.inc* file.

Each application should have its own *.inc* file that lists names of the Windows functions, data types, and constants it uses.

2.3.2 $WINDOWS Metacommand

Pascal applications must be compiled using the **$WINDOWS** metacommand.

2.3.3 Program Module

Pascal applications must be defined as Pascal modules, not programs. A program module can contain any number of Pascal procedure or function definitions. At least one function, the **WinMain** function, is required for any Windows application.

2.3.4 WinMain Function

The **WinMain** function is the entry point (starting point) of the application. This function contains statements and Windows function calls that create windows and read and dispatch input intended for the application.

The **WinMain** function, which must be declared with the **PUBLIC** attribute, has the following form:

```
function WinMain( hInstance     : HANDLE;
                  hPrevInstance: HANDLE;
                  lpCmdLine   : LPSTR;
                  nCmdShow       : INT ) : BOOL [ PUBLIC ];
```

2.3.5 Callback Functions

Callback functions are functions in an application that Windows calls to carry out specific tasks. An example is a window function that processes messages for an application's windows.

Windows expects callback functions to be public and to have the Windows prolog and epilog, therefore you must declare the functions with both the the WINDOWS and PUBLIC attributes. The WINDOWS attribute ensures that the Windows prolog and epilog code is included and that the correct data segment is used by the function when it executes. The following example shows the correct definition for a callback function:

```
function About ( hDlg    : HWND;
                 message: UNSIGNED;
                 wParam  : WORD;
                 lParam  : LONG ) : BOOL [ PUBLIC, Windows ];
```

All callback functions must be listed in the **EXPORTS** statement of the application's module definition file (see Chapter 4, "The Windows Linker: Link4," for an explanation of the module definition file). This identifies the function as a callback and permits Windows to insert the proper data-segment address in the function's prolog when it loads the application.

Local functions (functions used exclusively by the application and not called by Windows) do not require the Windows prolog and epilog and can use the ordinary Pascal calling conventions.

2.3.6 Windows and Pascal Libraries

If you write Windows applications in the Pascal language, you can link your applications with the following libraries:

paslibw.lib
pascal.lib

The *paslibw.lib* library contains Windows entry points for Windows applications. This library should be linked with all Pascal Windows applications. In most cases, it should appear as the first library in the **link4** command line.

The *pascal.lib* library is the standard run-time library for Pascal. This library should be linked with all Pascal Windows applications. It contains code for the procedures and functions implemented by the Pascal language.

Note

The *llibc.lib* library is actually the standard large-model run-time library for C. It is sometimes used with Pascal applications to resolve references made by functions in the *paslibw.lib* library, such as replacement code for Pascal memory functions.

2.3.7 Pascal Memory-Allocation Routines

The Pascal memory-allocation routines **ALLHQQ** and **ALLMQQ** used in standard Pascal applications should not be used in Windows applications developed in Pascal. In the current Pascal library *pascal.lib*, **ALLHQQ** calls **_nmalloc**, which in turns calls the Windows **LocalAlloc** function to create a fixed memory block in the application's local heap. Similarly, **ALLMQQ** calls **_fmalloc**, which calls the **GlobalAlloc** function to create a fixed global memory block. Since **LocalAlloc** and **GlobalAlloc** do not operate identically to the **ALLHQQ** and **ALLMQQ** functions in standard Pascal applications, their use is not recommended. Use Windows memory-manager functions for managing memory for Windows applications written in Pascal.

2.4 Assembly-Language Applications

Assembly-language Windows applications are highly structured assembly-language programs that use high-level-language calling conventions as well as Windows functions, data types, and programming conventions. Although you assemble assembly-language Windows programs using the Microsoft Macro Assembler, the goal is to generate object files that are similar to object files generated using the C compiler. The following is a list of guidelines designed to help you meet this goal and create assembly-language Windows applications:

1. Include the *cmacros.inc* file in the application source files. This file contains high-level-language macros that define the segments, programming models, function interfaces, and data types needed to create Windows applications. For a complete description of the C macros, see the *Microsoft Windows Programmer's Reference.*
2. Define the programming model, setting one of the options **memS**, **memM**, **memC**, or **memL** to 1. This option must be set before the statement including the *cmacros.inc* file.
3. Set the calling convention to Pascal by setting the **?PLM** option to 1. This option must be set before the statement including the *cmacros.inc* file. Pascal calling conventions are required only for functions that are called by Windows.

4. Create the application entry point, **WinMain**, and make sure it is declared a public function. It should have the following form:

```
cProc WinMain, <PUBLIC>, <si,di>
                parmW hInstance
                parmW hPrevInstance
                parmD lpCmdLine
                parmW nCmdShow
cBegin WinMain
                .
                .
                .
cEnd WinMain

sEnd
```

   The **WinMain** function should be defined within the standard code segment **CODE**.

5. Set the Windows prolog and epilog option **?WIN** to 1. This option must be set before the statement including the *cmacros.inc* file. This option is required only for callback functions (or for exported functions in Windows libraries).

6. Make sure your callback functions are declared public and far. They should have the following form:

```
cProc TestWndProc, <FAR,PUBLIC>, <si,di>
                parmW hWnd
                parmW message
                parmW wParam
                parmD lParam
cBegin TestWndProc
                .
                .
                .
cEnd TestWndProc
```

   Callback functions must be defined within a code segment.

7. Link your application with the appropriate C-language library for Windows and C run-time libraries. To link properly, you may need to add an external definition for the absolute symbol **__acrtused** in your application source file.

Chapter 3
Resource Compiler: Rc

3.1 Introduction

Applications for the Microsoft Windows operating environment typically use a variety of resources, such as icons, cursors, menus, and dialog boxes. These resources must be created and then defined in a file called the resource script file.

This chapter describes how to create a resource script file and how to compile your application's resources.

3.2 Building the Resource Script File

To define the resources for the application, you must create the resource script file. (You can create the script file by using an ordinary text editor.) The resource script file consists of one or more resource statements that define the resource name, type, and details.

The file also contains one or more directives, which are special statements that define actions to perform on the script file before it is compiled. Resource directives can assign values to names, include the contents of files, and control compilation of the script file. The resource directives are identical to the directives used in the C programming language. They are fully defined in the *Microsoft C Reference Manual.*

The resource script file has the extension *.rc.*

For a complete description of the resource statements, see the *Microsoft Windows Programmer's Reference.*

Example

```
#include "shapes.h"
shapes  cursor  shapes.cur
shapes  icon    shapes.ico
shapes  menu
begin
    popup "Shape"
        begin
            menuitem "Clear", CLEAR
            menuitem "Rectangle", TRECT
            menuitem "Triangle", TRIANGLE
            menuitem "Star", STAR
            menuitem "Ellipse", ELLIPSE
        end
end
```

In this example, the script file defines the resources for an application called "Shapes." The **cursor**, **icon**, and **menu** keywords specify the type of resource being described. The name of the resource appears on the left. The file containing the resource appears on the right. If the resource is defined in the script file (as is the **menu** resource, above), its definition follows the keyword and is enclosed in the keywords **begin** and **end**.

You are free to choose any names you like for the resources. However, the names must be letters and digits. You will use these names in your application to identify the resource you want to load. You can place several resources of the same type in a resource file, but no two resources of the same type can have the same name.

3.2.1 Directives in a Resource Script

The resource directives are special statements that define actions to perform on the script file before it is compiled. The directives can assign values to names, include the contents of files, and control compilation of the script file.

The resource directives are identical to the directives used in the C programming language. They are fully defined in the *Microsoft C Reference Manual.*

The script file can contain any number of the following directives:

#define	**#ifdef**
#elif	**#ifndef**
#else	**#include**
#endif	**#undef**
#if	

When using directives, the number sign (#) must appear in the first column of the line.

3.2.2 Using the #include Directive

Although resource statements and directives can be in any order in the resource script file, the **#include** directive has a slightly different action depending on what statements are placed before it. If an **#include** directive is placed before the first definition statement, the Windows resource compiler (**rc**) processes only the **#define** statements in the specified include file. If the **#include** directive is placed after the first definition statement, **rc** processes all statements in the include file. For example, in the following resource script file only the **#define** statements in the files *windows.h* and *mydefs.h* are processed. Other statements in these files are ignored. But all statements in the file *dlgs.rc* are processed.

```
#include "windows.h"
#include "mydefs.h"
myicon icon myicon.ico

#include "dlgs.rc"
```

3.2.3 Using the Rcinclude Keyword

Syntax

rcinclude *filename*

This keyword copies the contents of the file specified by *filename* into your resource script before **rc** processes the script. The **rcinclude** keyword differs from the **#include** directive in one important aspect: nothing in the file designated by **rcinclude** is ignored. In a file named by **#include**, anything other than definitions (**#define** statements) is ignored when the file is processed. Thus, you should use **rcinclude**, not **#include**, to put resources in files that you will name.

The *filename* parameter is an ASCII string, enclosed in double quotation marks, that specifies the DOS filename of the file to be included. A full pathname must be given if the file is not in the current directory or in the directory specified by the INCLUDE environment variable. The *filename* parameter is handled as a C string: two backslashes must be given wherever one is expected in the pathname (for example, "root\\sub"). A single forward slash (/) can be used instead of double backslashes (for example, "root/sub").

Example

```
#include "style.h"
hand icon hand.ico
name menu
    begin
        rcinclude menu.rc
    end
```

The *menu.rc* file contains the following:

```
menuitem "Pause", IDOK
```

3.3 Compiling Resources

You can compile your application's resources by using the Windows resource compiler, **rc**. The compiler reads a script file that contains a list of the resources you wish to compile and add to the resource file. The compiler automatically places the resources in the application's resource file after compiling them.

Syntax

rc [[**–r**]] *filename* [[*executable-file*]]

Parameter	Description
filename	The name of the script file that contains the names, types, filenames, and descriptions of the resources you want to add to the file.

executable-file The name of the executable file to put the resources into. If no executable file is given, the executable file having the same name as the script file is used.

The –**r** option directs **rc** to compile the resource file, then saves the result in a special binary resource file having the filename extension *.res*. When –**r** is specified, **rc** does not copy the compiled resource to the executable file.

Example

```
rc sample.rc
rc sample
```

Both of the commands in the example read the resource script file *sample.rc*, create a compiled resource file *sample.res*, and copy the resources to the executable file *sample.exe*. When a filename has no extension, *.rc* is assumed.

The following command creates the compiled resource file *sample.res*:

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

When –**r** is specified, **rc** does not copy the compiled resource to the executable file.

The following command searches the current directory for the compiled file *sample.res*. If the file is found, it copies the resources in it to the executable file *sample.exe*. If no *.res* file is found, **rc** terminates without searching for a resource script file.

```
rc sample.res
```

The following command compiles the script file *sample.rc* and copies the result to the executable file *run.exe*:

```
rc sample run.exe
```

Chapter 4
Windows Linker: Link4

4.1 Introduction

You create executable Windows applications and libraries by linking your compiled source files using the **link4** program. The **link4** program takes your compiled sources, a list of Windows and other libraries, and a module definition file (a text file containing information about your application or library) and creates a Windows executable file that you can load and run with Windows.

This chapter describes how to use **link4**, how to create module definitions files, and how to name the libraries to be used with your application library.

4.2 Creating Module Definition Files

A module definition file is an ordinary text file that defines the contents and system requirements of a Windows application or library. The file contains one or more module statements, each defining a specific attribute of the application or library, such as its module name, the number and type of program segments, and the number and names of exported and imported functions. Every application and library must have a module definition file.

You must create the file before linking the application. The file contains one or more definition statements. Each statement defines some aspect of the application or library, such as its module name and segment types. You can choose any filename for the file, but you must use the filename extension *.def*.

Sections 4.2.1 and 4.2.2 explain how to create the module definition files for applications and libraries. For a complete definition of the module definition statements, see the update to Chapter 6 of the *Microsoft Windows Programmer's Reference.*

4.2.1 Module Definitions for Applications

A module definition file for an application must contain a **NAME** statement defining the application's module name. This name is used by Windows to identify the application. Although this is the only required

statement in the module definition file, most files contain additional statements, such as the **DATA** and **CODE** statements, that define further aspects of the application.

The following example shows a typical module definition file for an application:

```
; Sample Module Definition File
NAME     Sample
DESCRIPTION      'Sample Window Application'

DATA     MULTIPLE          MOVEABLE
CODE     MOVEABLE

HEAPSIZE          4096
STACKSIZE         4096

EXPORTS
          SampleWndProc @1
```

In this example, the module name is "Sample." This module has **MULTIPLE** data segments (one for each instance). The data and code segments are **MOVEABLE**. The heap and stack sizes are 4096 bytes. The window function is named "SampleWndProc". The application imports no functions.

It is recommended that applications have at least 4096 bytes of stack space. Heap space is required if the application uses its local heap. The application's data segment must be **MULTIPLE**, since any application can be invoked more than once. Moveable code and data segments are recommended, since they allow Windows to take best advantage of memory.

The first line of the sample module definition file is a comment. A comment can appear on a line by itself or on the same line as a definition, as long as it appears after the definition. A comment must be preceded by a semicolon (;).

4.2.2 Module Definitions for Libraries

A module definition file for a library must contain a **LIBRARY** statement defining the library's module name. This name is used by Windows to identify the application. The file must also contain an **EXPORT** statement that lists the functions to be exported by the library. Functions in the library are not accessible if not listed.

The following example shows a typical module definition file for a library:

```
; Example Module Definition File
LIBRARY Example
DESCRIPTION     'Example Window Library'

DATA     SINGLE  MOVEABLE
CODE     MOVEABLE

HEAPSIZE          4096

EXPORTS
          ExampleInit @1
          ExampleStart @2
          ExampleEnd @3
          ExampleLoad @4
          ExampleSave @5
```

In this example, the module name is "Example." This module has **SINGLE** data segments (only one instance of a library is ever allowed). The data and code segments are **MOVEABLE**. The heap size is 4096 bytes. No **STACK** statement is given, which means the library will use the stack of the calling application. The exported functions are listed by name and ordinal number. These are the names or numbers that you can put in the **IMPORT** statement of an application's module definition file to indicate that the application calls the library.

4.3 Linking an Application

You can link the compiled application source files, the Windows library, and the module definition files by using **link4**, the Windows Linker. The **link4** utility combines the code and data of all application files with the appropriate code for any Windows functions called from within the application, and creates a new linked file that is in executable format.

4.3.1 Link4 Command

Syntax

link4 [[*options*]] *object-files*,[[*exe-file*]],[[*map-file*]],[[*lib-files*]],*def-file*

Parameter	Description
options	Is one or more keywords (described in Section 4.3.2) that direct **link4** to carry out special operations.
object-files	Specifies the filenames of compiled application source files. If your application has more than one compiled source file, you must name all of them when you link. This means you can give more than one *object-file* if necessary. Multiple filenames must be separated by spaces or the plus sign (+).
exe-file	Specifies the name you want the executable file to have.
map-file	Specifies the name you want the map file to have.
lib-files	Specifies the names of Windows or standard language libraries.
def-file	Specifies the filename of the module definition file. No application may have more than one *def-file*.

Commas are required to separate parameters in the command line.

Example

```
link4 sample/A:16,sample.exe,sample.map/map/li,slibw,sample.def
```

The command line in this example links the application object file *sample.obj* with the standard Windows library *slibw.lib*. This command creates the file *sample.exe* which has the module name, segments, and exported functions defined by the module definition file *sample.def*. It also creates the mapping file *sample.map*, which is used for symbolic debugging. The command searches the library file *slibw.lib* to resolve any external function calls made in the application files. It also searches any libraries in the object file's default library list.

Note

The **link4** utility uses default filename extensions if you do not explicitly provide them. Thus, in the preceding example, the filename *sample* is extended to *sample.obj*. Library names are extended with the *.lib* extension.

4.3.2 Link4 Options

The following list describes the **link4** options:

/alignment:*size*
This option directs **link4** to align segment data in the executable file along the boundaries specified by *size*. The *size* argument specifies a boundary size in bytes; for example, "alignment:16" indicates an alignment boundary of 16 bytes. The recommended alignment for Windows applications is 16 bytes. The *size* must be a power of 2; therefore, 2, 4, 8, 16, and so on are appropriate values. The default is 512 bytes. Minimum abbreviation: **/a**.

/help
This option directs **link4** to display a list of available options. Minimum abbreviation: **/h**.

/linenumbers
This option directs **link4** to copy line-number information from the object file to the map file. The option typically is used to prepare the map file for use with a source-level debugger, such as **symdeb**. Minimum abbreviation: **/li**.

/map
This option directs **link4** to copy information about each symbol in the application to the map file. The option typically is used to prepare a map file for use with a symbolic debugger, such as **symdeb**.

/nofarcalltrans

This option prevents the translation of far calls within the current segment. Without this option, far calls are translated into the following assembler statements:

```
    .
    .
    .
NOP
PUSH CS
NEAR CALL
    .
    .
    .
```

Minimum abbreviation: **/nof**.

/noignorecase

This option directs **link4** to preserve lowercase letters when matching symbols during linking. Minimum abbreviation: **/noi**.

/packcode[:*number*]

This option directs **link4** to pack contiguous logical or memory-code segments into one physical or file segment. The *number* parameter specifies the segment size limit in bytes. If no *number* is given, the default is 65,536. Minimum abbreviation: **/pac**.

/pause

This option directs **link4** to pause before copying the executable file to disk. Minimum abbreviation: **/pau**.

/segments:*number*

This option sets the maximum number of segments **link4** will process. The default is 128 segments. Minimum abbreviation: **/se**.

/stack:*size*

This option directs **link4** to set the stack size to *size* bytes. This option typically is used in place of the **stacksize** statement in the module definition file. Minimum abbreviation: **/st**.

/warnfixup

This option causes **link4** to display an error message when an offset fixup (relative to a logical segment) that is outside the physical segment occurs. Minimum abbreviation: **/w**

Note

There is an additional option, **/nodefaultlibrarysearch**, which causes **link4** to ignore default libraries. Some language compilers, such as the Microsoft C Compiler, add default library information to the object file. To ensure that the necessary library information is added to your application's object files, do not use this option.

4.4 Creating Import Libraries

You can create import libraries for Windows libraries by using the **implib** command. The command creates an import library file that can be specified in the **link4** command line with other libraries. Import libraries are required for all Windows libraries that can be linked dynamically. When you create a Windows library, you must create an import library to specify on the **link4** command line of the applications that use that library.

Syntax

implib *imp-lib-name mod-def-file*

Parameter	Description
mod-def-file	Specifies the name of the module definition file for the Windows library.
imp-lib-name	Specifies the name you want the new import library to have.

Example

```
implib mylib.lib mylib.def
```

This command creates the import library named *mylib.lib* from the module definition file *mylib.def*.

4.5 Examining Executable File Headers

You can use the **exehdr** command to determine whether an executable file is a Windows application or a library. The command also lets you find out which functions are exported or imported by a module, determine the amount of space allocated for a module's heap or stack, and determine the size and number of the segments a module contains.

The **exehdr** command has the following form:

exehdr *exe-filename*

The *exe-filename* is the name of any file with an *.exe* extension.

Example

```
exehdr hello.exe
```

This command displays the header for the executable file *hello.exe*. The format of this header is closely related to the statements contained in the application's module definition file.

Chapter 5
Symbolic Debugging Utility: Symdeb

5.1 Introduction

The Microsoft Symbolic Debug Utility (**symdeb**) is a debugging program that helps you test executable files. You can display and execute program code, set breakpoints that stop the execution of your program, examine and change values in memory, and debug programs that use the floating-point emulation conventions used by Microsoft languages.

The **symdeb** utility lets you refer to data and instructions by name rather than by address. The **symdeb** utility can access program locations through addresses, global symbols, or line-number references, making it easy to locate and debug specific sections of code.

You can debug C and Pascal programs at the source-file level as well as at the machine level. You can display the source statements of a program, the disassembled machine code of the program, or a combination of source statements and disassembled machine code.

The **symdeb** utility accepts source line numbers as arguments to commands for displaying and changing data, setting breakpoints, and tracing execution.

This chapter explains how to use **symdeb** to debug Windows applications. In particular, it describes how to do the following:

- Prepare symbol files for an application
- Set up the debugging terminal
- Start **symdeb** with Windows
- Interpret **symdeb**'s allocation messages
- Display the application's code and view its source file
- Set breakpoints and interpret backtraces
- Work with multiple instances of the same application
- Kill an application and quit **symdeb**

Note

If you have both a standard and a debugging version of Windows, **symdeb** may retrieve the incorrect version of certain files for use in debugging. To correct this problem, include in your DOS PATH variable the name of the development directory you created when installing the development kit. (If the directory with the standard version of Windows is in your PATH, be sure the development directory is listed first.) Then **symdeb** automatically will retrieve files from the debugging version of Windows.

5.2 Preparing Symbol Files

Windows applications are difficult to debug without symbolic information about Windows and the application. To take advantage of **symdeb**'s symbolic features, you must first prepare a symbol file that **symdeb** can use.

The steps for setting up a symbol file depend on the method used to create the program. The following sections describe those steps for applications written in C, Pascal, or assembly language.

5.2.1 Mapsym Program

The **mapsym** program creates symbol files for symbolic debugging. The program converts the contents of an application's symbol map (*.map*) file into a form suitable for loading with **symdeb**, copying the result to a symbol (*.sym*) file.

Syntax

mapsym [[{/|–}**l**]] [[{/|–}**n**]] *mapfilename*

Parameter	Description
mapfilename	The filename for a symbol map file that was created during linking. If you do not give a filename extension, *.map* is assumed. If you do not give a full pathname, the current directory and drive is assumed. The **mapsym** program creates a new symbol file having the same name as the map file but with the *.sym* extension.
/l	An option that directs **mapsym** to display information about the conversion on the screen. The information includes the names of groups defined in the program, the program start address, the number of segments, and the number of symbols per segment.
/n	An option that directs **mapsym** to ignore line-number information in the map file. The resulting symbol file contains no line-number information.

Example

```
mapsym /l file.map
```

In this example, **mapsym** uses the symbol information in *file.map* to create *file.sym* on the current drive and directory. Information about the conversion will be sent to the screen.

Note

The **mapsym** program always places the new symbol file in the current drive and directory.

To create a map file for **mapsym** input, you must specify the **/map** option when linking. To add line-number information to the map file, you specify the appropriate option when compiling, and specify the **/linenumbers** option when linking.

The **mapsym** program can process up to 10,000 symbols for each segment in the application.

5.2.2 Symbols with C-Language Applications

To prepare a symbol file for an application written in the C language, follow these steps:

1. Compile your source file using the **–Zd** option to produce line numbers in the object file. Debugging is easier if you disable the compiler's optimization.
2. Link the object file to produce an executable version of the program. Specify a map filename in the linker's command line and give the **/map** and **/linenumbers** options. Make sure the map filename is the same as the application's module name given in the module definition file.
3. Use the **mapsym** program to produce a symbol file.

Example

```
cc -d -c -Asnw -Gsw -Os -Zdpe test.c
link4 test,test,test/map/li,slibw, test
mapsym test
```

5.2.3 Symbols with Pascal Applications

To prepare a symbol file for an application written in the Pascal language, follow these steps:

1. Compile your source file using the **/l** option for line-number information.
2. Link the object file to produce an executable version of the application. Specify a map filename in the linker's command line and give the **/map** and **/linenumbers** options. Make sure the map filename is the same as the application's module name given in the module definition file.
3. Use the **mapsym** program to produce a symbol file.

Example

```
pas1 /l test.pas;
pas2
pas3
link4 test,test,test/map/li, paslibw pascal llibc,test
mapsym test
```

5.2.4 Symbols with Assembly-Language Applications

To prepare symbol files for Windows applications written in assembly language, follow these steps:

1. Make sure that all symbols you may want to use with **symdeb** are declared public. Segment and group names should not be declared public. They automatically are available for debugging.
2. Assemble your source file.
3. Link the object file to produce an executable version of the application. Specify a map filename in the linker's command line and give the **/map** option. Make sure the map filename is the same as the application's module name given in the module definition file.
4. Use the **mapsym** program to create a symbol file.

Example

```
masm test;
link4 test,test,test/map,slibw slibc libh,test
mapsym test
```

5.3 Setting Up the Debugging Terminal

While it is running, Windows takes complete control of the system console, making debugging through the console impossible. To debug Windows applications, you either can set up a remote terminal, connected through the computer's serial port, or set up a secondary monochrome display adapter and monitor.

5.3.1 Setting Up a Remote Terminal

To set up a remote terminal for debugging, follow these steps:

1. Select a serial port on your computer and connect a terminal to it.
2. Use the DOS **mode** command to set the baud rate and line protocol of the serial line to correct values for use with the terminal. Line protocol includes the number of stop bits, type of parity checking, and number of transmission bits used by the terminal.
3. When you start **symdeb**, redirect **symdeb**'s input and output to the remote terminal using the = command to specify a communications port. For example, the command "=com2" redirects all subsequent **symdeb** command input and output to *com2*.

Note

Debugging through a remote terminal disables the normal function of the CONTROL-S keys. These keys cannot be used while debugging Windows applications.

5.3.2 Setting Up a Secondary Monitor

To set up a secondary monitor for debugging, follow these steps:

1. Install a secondary monochrome display adapter in a free slot of your computer and connect the monochrome monitor to it.
2. Set the secondary display adapter switches to the appropriate settings. Follow the display adapter and computer manufacturer's recommendations.
3. When you start **symdeb**, use the **/m** option to redirect **symdeb** output to the secondary monitor.

Note

When the **/m** option is given, **symdeb** redirects output to the secondary monitor, but continues to use the system keyboard for input until the application being debugged is started. While the application is running, **symdeb** yields complete control of the keyboard to the application. As soon as the application reaches a breakpoint or terminates, **symdeb** reclaims the keyboard and permits user input again.

5.4 Starting Symdeb with Windows

To start **symdeb** with Windows, enter the following **symdeb** command line at the DOS command prompt:

symdeb [[*options*]] [[*symbolfiles*]] **win.com** [[*arguments*]]

The *options* are one or more **symdeb** options. The *symbolfiles* are the names of symbol files. The *arguments* are parameters that you want to pass to *win.com*.

Once started, **symdeb** displays a startup message followed by the **symdeb** command prompt (–). When you see the prompt you can enter **symdeb** commands. The **symdeb** commands are described in Section 5.9.

Sections 5.4.1 and 5.4.2 describe the elements of the **symdeb** command line.

5.4.1 Symdeb Options

You can specify one or more **symdeb** options in the command line. The **symdeb** options control the operation of **symdeb** while debugging Windows applications. Options must appear before *win.com* on the command line so that **symdeb** will not interpret them as program arguments.

The **symdeb** utility has the following command line options:

Option	Meaning
/m	Redirects **symdeb** output to a secondary monochrome monitor and permits debugging of Windows applications without redirecting input and output to a remote terminal. The **symdeb** utility assumes that the necessary display adapter and monitor are installed.
/x	Disables the "more" feature. Unless this option is specified, **symdeb** automatically stops lengthy output and does not continue the display until the user presses a key. The **symdeb** utility stops the output after displaying enough lines to fill the screen, then prompts the user to continue by displaying the message "[more]". If this option is specified, **symdeb** continues to display output until the command is completely executed.
/w*number*	Sets the memory-allocation reporting level. The reporting level defines what kind of memory allocation and movement messages **symdeb** is to display when Windows loads and moves program segments. The *number* specifies the reporting level and can be 0, 1, 2, or 3. Level 0 specifies no reporting. Level 1, the default level if the **/w** option is not given, generates allocation messages only. Level 2 generates movement messages only. Level 3 generates both allocation and movement messages. See Section 5.7 for more information about allocation messages.
/@*filename*	Directs **symdeb** to load macro definitions from the file specified by *filename*. Macro definitions define the meaning of **symdeb**'s ten macro commands. The given file must contain one or more macro definitions in the following form: **m***number*=*command-string* The *number* specifies the macro and *command-string* is one or more **symdeb** commands separated by semicolons (;).

/n

Permits use of non-maskable interrupts on non-IBM computers. To use non-maskable interrupts, you must have a system that is equipped with the proper hardware, such as the following products:

- IBM Professional Debugging Facility
- Software Probe (Atron Corporation)

The **symdeb** utility requires only the hardware provided with these products; no additional software is needed. If you are using one of these products with a non-IBM system, you must use the **/n** option to take advantage of the break capability. Using a non-maskable-interrupt break system is more reliable than using the interactive break key because you can always stop program execution regardless of the state of interrupts and other conditions.

/i[**bm**]

Directs **symdeb** to use features available on IBM-compatible computers. The option is not necessary if you have an IBM Personal Computer, because **symdeb** automatically checks the hardware on startup. If **symdeb** does not find that the hardware is an IBM Personal Computer, it assumes that the hardware is a generic MS-DOS machine. Without the option, **symdeb** cannot take advantage of special hardware features such as the 8259 Interrupt Controller, the IBM-style video display, and other capabilities of the IBM basic input and output system (BIOS).

/f*filename*

Prevents association of the named symbol file with the executable file that has the same name. This option is rarely used and is not recommended for debugging Windows applications.

/commands

Directs **symdeb** to execute commands in the *commands* list immediately after starting. Commands in the list must be separated with semicolons and the entire list must be enclosed in double quotation marks. The / is required.

Note

The option designator can be either a slash (/) or a dash (–), and the option letter can be given with either uppercase or lowercase letters.

Files containing a dash in the filename must be renamed before use with **symdeb**. Otherwise, **symdeb** will interpret the dash as an option designator.

5.4.2 Specifying Symbol Files

To debug a Windows application symbolically, you should load symbol files for the following items:

- The application
- Windows kernel, user, and GDI libraries
- Other Windows libraries used by the application

The symbol file for the application is required. The symbol files for the Windows libraries are optional, but recommended. They are helpful when trying to trace calls made to routines that are not in the application or to trace window messages.

You must give the complete filename and extension when naming a symbol file. If the symbol file is not in the current directory, you must supply a full pathname. All symbol files must be specified before the *win.com* file.

You should always name the application symbol file before any other symbol files. This ensures that the application's symbol file is open and you can access the application's symbols when you first start **symdeb**.

Example

```
symdeb \app\test.sym user.sym gdi.sym \app\testlib.sym win.com
```

Note

The Windows kernel, user, and GDI library symbol files, *kernel.sym*, *user.sym*, and *gdi.sym*, are provided as part of the Microsoft Windows Software Development Kit.

You can create symbol files for other Windows libraries by using the same methods you used to create application symbol files.

5.4.3 Passing the Application to Windows

You can pass the name of your application to Windows by placing the full pathname on the **symdeb** command line immediately after the *win.com* filename. Windows receives the name as an argument when you start *win.com* from within **symdeb**. Windows uses the name to load and run the application.

Example

```
symdeb \app\test.sym win.com \app\test.exe
```

If you do not supply your application's name as an argument, you can load and start your application by starting *win.com* and using the MS-DOS Executive to load the application.

5.4.4 Symdeb Keys

The **symdeb** utility provides a number of special keys for controlling input and output and program execution. The following is a list of these keys:

Key	Action
SCROLL-LOCK	Suspends and restores **symdeb** output. The key typically is used to stop temporarily the output of lengthy displays. To suspend output, press the SCROLL-LOCK key. To restore output, press the key again.

SYS-REQ	Generates an immediate breakpoint that halts program execution and returns control to **symdeb**. (Available on the IBM Personal Computer AT only).
CONTROL-C	Cancels the current **symdeb** command. This key does apply to commands that pass execution control to the application being debugged.

5.5 Working with Symbol Maps

Symbol files that **symdeb** has loaded for debugging are called symbol maps. The **symdeb** utility lets you examine symbol maps and use the symbols in the maps to set breakpoints and display variables and functions.

Although symbol maps are in memory, **symdeb** allows access to only one symbol map at a time. You can display a list of symbol maps at any time, but to display or use the symbols in a map, you first must open that map.

Note

The **symdeb** utility requires that the filename part of the application's *.sym* file be the same as the application's module name (specified in the application's module definition file). If these names are not identical, **symdeb** will not be able to determine the correct segment addresses for symbols in the application.

5.5.1 Listing the Symbol Maps

You can display a list of the symbol maps by using the **x** command with the asterisk (*) argument. The command displays the names of all maps in memory, the name of each segment belonging to a map, and the 16-bit paragraph address of each segment. (The **x** command without an argument displays only the open map.)

Example

```
- x *
[ 0000 TEST ]
          [ 0001 IGROUP ]
            0002 DGROUP
  0000 TESTLIB
            0001 _TEXT
            0002 DGROUP
```

The open map name is enclosed in brackets ([ ]). The active segment in the map is also enclosed in brackets. Segment addresses appear immediately before the segment names.

Note

Symdeb does not display a segment's actual segment address until the code or data corresponding to that segment has been loaded. If you list the symbol maps before loading an application, **symdeb** displays low-memory addresses as a warning that the segments are not yet in memory.

Once an application is loaded, **symdeb** appends a number to the end of the data segment name in the symbol map. This number shows which instance of the application the data segment belongs to. If you load multiple instances of an application, **symdeb** adds a new data segment name to the symbol map for that application.

In the following example, **symdeb** places parentheses around the active data segment to show which instance of the application currently is running.

```
[ 0000 TEST ]
          [ 88F0 IGROUP ]
          ( 87E0 DGROUP  )
            8944 DGROUP1
```

5.5.2 Opening a Symbol Map

To access the symbols in a symbol map, you must open the symbol map using the **xo** command. For example, to open the symbol map named *test*, you would type the following:

```
-xo test!
```

The **symdeb** utility opens the symbol map and lets you examine and use symbols from the map.

You can use the **xo** command to open a different symbol map at any time. Since only one symbol map can be open at a time, the previous symbol map is closed.

Note

When you load multiple symbol maps, **symdeb** automatically opens the first one loaded.

5.5.3 Display Symbols

You can use the **x?** command to display the symbols in the open symbol map. The command lists each symbol by name and also gives the symbol's address offset. For example, to display the symbol "testwndproc," you would type the following:

```
-x? testwndproc
[ 88E0 IGROUP ]
   005A TESTWNDPROC
```

The command displays the name and address of the segment to which the symbol belongs. The symbol's absolute address can be computed using the segment's address and the symbol's offset. In the preceding example, the function "testwndproc" is in the segment IGROUP at address 88E0:005A.

If the symbol is an external symbol (for example, a function or variable defined outside of the application), no group name is given and the offset is always zero, as shown in the following example:

```
-x? showwindow
0000 SHOWWINDOW
```

You can use the asterisk (*) as a wildcard character with the **x** command to display more than one symbol at a time. For example, the following command displays all symbols in the IGROUP segment:

```
-x? igroup:*
```

The following command displays all symbols in the DGROUP segment that begin with an underscore (_):

```
-x? dgroup:_*
```

5.6 Starting the Application

You can start the application by using the **g** command. The command directs **symdeb** to pass execution control to the program at the current code address. (Immediately after starting **symdeb** with Windows, the current code address is the start address of the *win.com* file.)

If you have supplied your application's filename as a *win.com* argument on the **symdeb** command line, *win.com* starts your application automatically. Otherwise, it starts the MS-DOS Executive, which you can use to load and run your application.

5.7 Allocation Messages

The **symdeb** utility displays memory allocation messages to show that Windows has created, freed, or moved memory blocks. The messages are intended to help you locate your application's program code and data in memory. The messages also can be used to see the effect of the application on Windows memory management. The **symdeb** utility actually displays messages only if the memory-allocation reporting level is set to an appropriate value (see the **/w** option in the preceding Section 5.4.1).

When Windows allocates a new block of memory and the reporting level is 1 or 3, **symdeb** displays a message of the following form:

module-name! *segment-name*=*segment-address*

The *module-name* is the name of the application or library to receive the allocated memory. The *segment-name* is the name of the code or data segment within the application or library that will occupy the memory block. The *segment-address* is the 16-bit paragraph address of the memory block.

When Windows moves a block of memory and the reporting level is 2 or 3, **symdeb** displays a message of the following form:

old-address `moved to` *new-address*

The *old-address* and *new-address* are the old and new 16-bit paragraph addresses of the memory block.

When Windows frees a block of memory and the reporting level is 1 or 3, **symdeb** displays a message of the following form:

segment-address `freed`

The *segment-address* is the 16-bit paragraph address of the block to be freed.

Example

```
TEST!IGROUP=886F
TEST!DGROUP=8799
GDI!Code=1C32
8344 moved to 8230
7C12 freed
```

5.7.1 Setting Breakpoints with Symbols

You can use the **bp** command and symbols to set breakpoints in your application code even before loading the application. The **bp** command uses the symbol to compute the instruction address at which to break execution. If the application has not been loaded, **symdeb** sets a virtual breakpoint. A virtual breakpoint has no effect on execution until the application actually is loaded. Once an application is loaded, **symdeb** computes the actual code addresses of all virtual breakpoints and enables the breakpoints.

For example, to set a breakpoint at the application's **WinMain** function, you would type the following:

```
-bp winmain
```

After you set the breakpoint, the application breaks and returns control to **symdeb** when this address is encountered.

Note

If you do not set breakpoints before starting the application, you can use an interrupt key to break execution (see section 5.4.4 for more information on **symdeb** keys).

5.7.2 Displaying Variables

You can use the **d** command to display the content of the application's static variables. The command takes the variable's symbol as an argument and computes the variable's address using the address of the variable's segment and its offset. The symbol map containing the symbol must be open.

Example

```
-dw hinstance
8882:0010 0143 0000 0000 0000 0000 0000 0000 0000
```

When there are multiple instances of the application being debugged, **symdeb** uses the address of the active data segment to compute a variable's address. To display a variable in another instance, you must supply an absolute segment address. For example, to display the value of *hInstance* in the first instance, the 16-bit paragraph address of the first DGROUP segment must be given explicitly, as shown in the following example:

```
-x
[ 0000 TEST ]
        [ 8A12 IGROUP ]
          89A0 DGROUP
        ( 8882 DGROUP1 )
-dw 89A0:hinstance
88A0:0010 0235 0000 0000 0000 0000 0000 0000 0000
```

5.7.3 Displaying Application Source Statements

You can display the source statements of an application by using the **v**, **s+**, and **s&** commands. The **v** command displays the source lines of the application beginning with the source line corresponding to the current code address (**cs:ip**). The **s+** command directs **symdeb** to display source lines whenever the **u** command is used. The **s&** command directs **symdeb** to display both source lines and unassembled code whenever the **u** command is used. For more information on these commands, see Section 5.9.

Note

The **v**, **s+**, and **s&** commands require that line-number information has been copied to the application's symbol file. If a symbol file does not contain line-number information, these commands have no effect.

If the application source file is not in the current directory, or the file does not have the same name as the symbol file, **symdeb** prompts for the file's correct name and/or pathname.

5.8 Quitting Symdeb

You can terminate **symdeb** at any time by using the **q** command to return to the DOS prompt. Before quitting **symdeb**, however, you may need to end the current Windows session and restore the console display to its normal display modes.

Follow these general rules:

- If you have not started Windows, use the **q** command to quit.
- If you have started Windows and it is still operational, open the MS-DOS Executive window and choose the Close command from its System menu, then use the **q** command.

Important

When Windows terminates as a result of a fatal exit, **symdeb** displays a fatal exit message and returns the **symdeb** prompt. Do not attempt to restart Windows or quit **symdeb**. You must reboot the system to continue.

5.9 Symdeb Commands

This section contains a complete listing of commands that can be used with **symdeb**. It also describes the arguments and expressions used with **symdeb** commands, as well as predefined names used as register and register flag names. The following table is a summary of **symdeb** command syntax and purpose.

Table 5.1
Symdeb Commands

Syntax	Meaning
a [[*address*]]	Assemble
bc [[*id-list*]]	Clear breakpoint(s)
bd [[*id-list*]]	Disable breakpoint(s)
be [[*id-list*]]	Enable breakpoint(s)
bl	List breakpoint(s)
bp[[*id*]] *address* [[*value*]] [[*command-string*]]	Set breakpoint
c *range address*	Compare
d [[*range*]]	Dump memory using previous type
da [[*range*]]	Dump memory ASCII
db [[*range*]]	Dump memory bytes
dd [[*range*]]	Dump memory double-words
dg	Display global heap
dh	Display local heap for current **ds**
dl [[*range*]]	Dump memory long floating point
dq	Display task queue
ds [[*range*]]	Dump memory short floating point
dt [[*range*]]	Dump memory ten-byte reals
dw [[*range*]]	Dump memory words
e *address* [[*list*]]	Enter using previous type
ea *address* [[*list*]]	Enter ASCII
eb *address* [[*list*]]	Enter bytes
ed *address* [[*list*]]	Enter double-words
el *address* [[*list*]]	Enter long floating point
es *address* [[*list*]]	Enter short floating point
et *address* [[*list*]]	Enter ten-byte reals
ew *address* [[*list*]]	Enter words
f *range list*	Fill
g [[=*address* [[*address*]]...]]	Go
h *value value*	Add hexadecimal
i *value*	Input from port
k [[*value*]]	Backtrace stack

Table 5.1 *(continued)*

Syntax	Meaning
kt *pdb* [[*value*]]	Backtrace task
l [[*address* [[*drive record count*]]]]	Load
m *range address*	Move
m*id*[[=*command-string*]]	Define or execute macro
n *filename* [[*filename...*]]	Set name
o *value byte*	Output to port
p [[=*address*]] [[*value*]]	Trace program instruction or call
q	Quit
r [[*register*]] [[[=] *value*]]	Register
s *range list*	Search
s–	Set machine debugging only
s&	Set machine and source debugging
s+	Set source debugging only
t [[=*address*]] [[*value*]]	Trace program instruction
u [[*range*]]	Display unassembled instructions
v [[*range*]]	View source lines
w [[*address* [[*drive record count*]]]]	Write to disk
x [[**?**]] *symbol*	Examine symbols(s)
xo *symbol*	Open map/segment
z *symbol value*	Set symbol value
? *expression*	Compute and display expression
.	Display current source line
< *filename*	Redirect **symdeb** input
> *filename*	Redirect **symdeb** output
=*filename*	Redirect **symdeb** input and output
{ *filename*	Redirect program input
} *filename*	Redirect program output
~ *filename*	Redirect program input and output
! [[*dos-command*]]	Shell escape
* *string*	Comment

Any combination of uppercase and lowercase letters may be used in commands and arguments. If a command takes two or more parameters, separate them with a single comma (,) or one or more spaces.

Examples

```
ds _avg L 10
U .22
f DS:100,110 ff,fe,01,00
```

5.9.1 Command Arguments

Command arguments are numbers, symbols, line numbers, or expressions used to specify addresses or values to be used by **symdeb** commands. The following is a list of argument syntax and meaning:

Argument	Description
address	Specifies absolute, relative, or symbolic address of a variable or function. The **symdeb** utility permits a wide variety of address types. See Section 5.9.2 for a complete description of address arguments.
byte	Specifies a *value* argument representing a byte value. It must be in the range 0 to 255.
command-string	Specifies one or more **symdeb** commands. If more than one command is given, each must be separated by semicolons (;).
count	Specifies a *value* argument representing the number of disk records to read or write.
dos-command	Specifies a DOS command.
drive	Specifies a *value* argument representing a disk drive. Drives are numbered zero for A, 1 for B, 2 for C, and so on.
expression	Specifies a combination of arguments and operators that represents a single value or address. See Section 5.9.3 for a list and explanation of expression operators.
filename	Specifies the name of a file or a device. The filename must follow the DOS file-naming conventions.

id

Specifies a decimal number representing a breakpoint or macro identifier. The number must be in the range 0 to 9.

id-list

Specifies one or more unique decimal numbers representing a list of breakpoint identifiers. The numbers must be in the range 0 to 9. If more than one number is given, each must be separated using spaces or commas. The wildcard character (*) can be used to specify all breakpoints.

list

Specifies one or more *value* arguments. The values must be in the range 0 to 65,535. If more than one value is given, they must be separated using spaces or commas.

A list can also be specified as a list of ASCII values. The list can contain any combination of characters and must be enclosed in either single or double quotation marks. If the enclosing mark appears within the list, it must be given twice.

range

Specifies an address range. Address ranges have two forms: a start and end address pair and start address and object count. The first form consists of two address arguments, the first specifying the start address and the second specifying the end address. The second form consists of an address argument, the letter **l**, and a value argument. The address specifies the starting address, the value specifies the number of objects after the address to examine or display. The size of an object depends on the command. If a command requires a range but only a start address is given in the command, the command assumes the range to have an object count of 128. This default count does not apply to commands that require a range followed immediately by a value or address argument.

record

Specifies a *value* argument representing the first disk record to be read or written to.

register

Specifies the name of a CPU register. It can be any one of the following:

ax	**cx**	**es**	**si**
bp	**di**	**f**	**sp**
bx	**ds**	**ip**	**ss**
cs	**dx**	**pc**	

The names **ip** and **pc** name the same register: the instruction pointer. The name **f** is a special name for the flags register. Each flag within the flags register has a unique name based on its value. These names can be used to set or clear the flag. Table 5.2 lists the flag names:

Table 5.2
Flag Values

Flag	Set	Clear
Overflow	**ov**	**nv**
Direction	**dn** (decrement)	**up** (increment)
Interrupt	**ei** (enabled)	**di** (disabled)
Sign	**ng** (negative)	**pl** (positive)
Zero	**zr**	**nz**
Auxiliary Carry	**ac**	**na**
Parity	**pe** (even)	**po** (odd)
Carry	**cy**	**nc**

symbol

Identifies the address of a variable, function, or segment. A symbol consists of one or more characters, but always begins with a letter, underscore (_), question mark (?), at sign (@), or dollar sign ($). Symbols are only available when the *.sym* file that defines their names and values has been loaded. Any combination of uppercase and lowercase letters can be used; **symdeb** is not case-sensitive. For some commands, the wildcard character (*) may be used as part of a symbol to represent any combination of characters.

pdb	Specifies a *value* argument representing the segment address of a program descriptor block.
value	Specifies an integer number in binary, octal, decimal, or hexadecimal format. A value consists of one or more digits optionally followed by a radix: Y for binary, O or Q for octal, T for decimal, or H for hexadecimal. If no radix is given, hexadecimal is assumed. **Symdeb** truncates leading digits if the number is greater than 65535. Leading zeroes, if any, are ignored. Hexadecimal numbers have precedence over symbols. Thus FAA is a number.

5.9.2 Address Arguments

Address arguments specify the location of variables and functions. The following list explains the syntax and meaning of the various addresses used in **symdeb**:

Syntax	Meaning
segment:*offset*	Specifies an absolute address. A full address has both a *segment* address and an *offset*, separated by a colon (:). A partial address is just an *offset*. In both cases, the *segment* or *offset* can be any number, register name, or symbol. If no *segment* is given, the **a**, **g**, **l**, **p**, **t**, **u**, and **w** commands use the **cs** register for the default segment address. All other commands use **ds**.
name{+ ¦ −}*offset*	Specifies a symbol-relative address. The *name* can be any symbol. The *offset* specifies the offset in bytes. The address can be specified as a positive (+) or negative (–) offset.
.{+ ¦ −}*number*	Specifies a relative line number. The *number* is an offset (in lines) from the current source line to the new line. If + is given, the new line is closer to the end of the source file. If – is given, the new line is closer to the beginning.

.[[*filename*:]]*number*	Specifies an absolute line number. If *filename* is given, the specified line is assumed to be in the source file corresponding to the symbol file identified by *filename*. If no filename is given, the current instruction address (the current values of the **cs** and **ip** registers) determines which source file contains the line.
.*symbol*[[{+ | −}*number*]]	Specifies a symbolic line number. The *symbol* can be any instruction or procedure label. If *number* is given, the *number* is an offset (in lines) from the given label or procedure name to the new line. If + is given, the new line is closer to the end of the source file. If – is given, the new line is closer to the beginning.

Note

Line numbers can be used only with programs developed with compilers that copy line-number data to the object file.

5.9.3 Expressions

An expression is a combination of arguments and operators that evaluates to an 8-, 16-, or 32-bit value. Expressions can be used as values in any command.

An expression can combine any symbol, number, or address with any of the unary and binary operators in the following Tables 5.3 and 5.4.

Table 5.3

Unary Operators

Operator	Meaning	Precedence
+	Unary plus	Highest
–	Unary minus	
not	1's complement	
seg	Segment address of operand	
off	Address offset of operand	
by	Low-order byte from given address	
wo	Low-order word from given address	
dw	Double-word from given address	
poi	Pointer from given address (same as **dw**)	
port	One byte from given port	
wport	Word from given port	Lowest

Table 5.4

Binary Operators

Operator	Meaning	Precedence
*	Multiplication	Highest
/	Integer division	
mod	Modulus	
:	Segment override	
+	Addition	
–	Subtraction	
and	Bitwise Boolean AND	
xor	Bitwise Boolean exclusive OR	
or	Bitwise Boolean OR	Lowest

Unary address operators assume **ds** as the default segment for addresses. Expressions are evaluated in order of operator precedence. If adjacent operators have equal precedence, the expression is evaluated from left to right. Parentheses can be used to override this order.

Examples

```
SEG 0001:0002                ; Equals 1
OFF 0001:0002                ; Equals 2
4+2*3                        ; Equals 10 (0Ah)
4+(2*3)                      ; Equals 10 (0Ah)
(4+2)*3                      ; Equals 18 (12h)
```

5.9.4 Assemble Command

Syntax

a[*address*]

This command assembles instruction mnemonics and places the resulting instruction codes into memory at *address*. If no *address* is given, the assembly starts at the address given by the current values of the **cs** and **ip** registers. To assemble a new instruction, type the desired mnemonic and press the ENTER key. To terminate assembly, press the ENTER key only. There are the following assembly rules:

- Use **retf** for the far return mnemonic.
- Use **movsb** or **movsw** for string-manipulation mnemonics.
- Use the **near** or **far** prefix with labels to override default distance. The **ne** abbreviation stands for **near**.
- Use the **word ptr** or **byte ptr** prefix with destination operands to specify size. The **wo** abbreviation stands for **word ptr**; **by** for **byte ptr**.
- Use square brackets around constant operands to specify absolute memory addresses. Constants without brackets are treated as constants.
- Use the **db** mnemonics to assemble byte values or ASCII strings directly into memory.
- Use 8087 or 80287 instructions only if your system has these math coprocessors.

80286 protected-mode mnemonics cannot be assembled.

5.9.5 Breakpoint Clear Command

Syntax

bc *id-list*

This command permanently removes one or more previously set breakpoints. If *id-list* is given, the command removes the breakpoints named in the list. The *id-list* can be any combination of integer values from 0 to 9. If * is given, the command removes all breakpoints.

5.9.6 Breakpoint Disable Command

Syntax

bd *id-list*

This command disables, but does not delete, one or more breakpoints. If *id-list* is given, the command disables the breakpoints named in the list. The *id-list* can be any combination of integer values from 0 to 9. If * is given, the command disables all breakpoints.

5.9.7 Breakpoint Enable Command

Syntax

be *id-list*

This command restores one or more breakpoints that were temporarily disabled by a **bd** command. If *id-list* is given, the command enables the breakpoints named in the list. The *id-list* can be any combination of integer values from 0 to 9. If * is given, the command enables all breakpoints.

5.9.8 Breakpoint List Command

Syntax

bl

This command lists current information about all breakpoints. The command displays the breakpoint number, the enabled status, the address of the breakpoint, the number of passes remaining, and the initial number of passes (in parentheses). The enable status can be enabled (e), disabled (d), or virtual (v). A virtual breakpoint is a breakpoint set at a symbol whose *.exe* file has not yet been loaded.

5.9.9 Breakpoint Set Command

Syntax

bp[*id*] *address* [*value*] [*command-string*]

This command creates a "sticky" breakpoint at the given *address*. Sticky breakpoints stop execution and display the current values of all registers and flags. Sticky breakpoints remain in the program until removed using the **bc** command, or temporarily disabled using the **bd** command. The **symdeb** utility allows up to ten sticky breakpoints (0 through 9). The *id* specifies which breakpoint is to be created. If no *id* is given, the first available breakpoint number is used. The *address* can be any valid instruction address (that is, it must be the first byte of an instruction). The *value* specifies the number of times the breakpoint is to be ignored before being taken. It can be any 16-bit value. The *command-string* is an optional list of commands to be executed each time the breakpoint is taken. Each **symdeb** command in the list can include parameters and is separated from the next command by a semicolon.

5.9.10 Compare Command

Syntax

c *range address*

This command compares the bytes in the memory locations specified by *range* with the corresponding bytes in the memory locations beginning at *address*. If all corresponding bytes match, the command displays its prompt and waits for the next command. If one or more corresponding bytes do not match, the command displays each pair of mismatched bytes.

5.9.11 Dump Command

Syntax

d [[*range*]]

This command displays the contents of memory in the given *range*. The command displays data in the same format as the most recent dump command. (Dump commands include **d**, **da**, **db**, **dd**, **dg**, **dh**, **dl**, **dq**, **ds**, **dt**, and **dw**). If no *range* is given and no previous dump command has been used, the command displays bytes starting from **ds:ip**.

5.9.12 Dump ASCII Command

Syntax

da [[*range*]]

This command displays the ASCII characters in the *range*. Each line displays up to 48 characters. The display continues until the first zero bytes or until all characters in the *range* have been shown. Nonprintable characters, such as carriage returns and line feeds, are displayed as a period (.).

5.9.13 Dump Bytes Command

Syntax

db [[*range*]]

This command displays the hexadecimal and ASCII values of the bytes in the *range*. Each display line shows the address of the first byte in the line, followed by up to 16 hexadecimal byte values. The byte values immediately are followed by the corresponding ASCII values. The eighth and ninth hexadecimal values are separated by a hyphen (–). Nonprintable ASCII values are displayed as periods (.).

5.9.14 Dump Double-words Command

Syntax

dd [[*range*]]

This command displays the hexadecimal values of the double-words (4-byte values) in the *range*. Each display line shows the address of the first double-word in the line and up to four hexadecimal double-word values.

5.9.15 Display Global Heap Command

Syntax

dg

This command displays a list of the global memory objects in the global heap. The list has the following form:

segment-address: *size segment-type owner*

The *segment-address* specifies the segment address of the first byte of the memory object. The *size* specifies the size in paragraphs (multiples of 16

bytes) of the object. The *segment-type* specifies the type of object. It can be any one of the following:

Segment Type	Meaning
CODE	Segment contains program code.
DATA	Segment contains program data and possible stack and local heap.
PRIV	Segment contains private data.
FREE	Segment belongs to pool of free memory objects ready for allocation by an application.
SENTINAL	Segment marks the beginning or end of the global heap.

The *owner* specifies the module name of the application or library that allocated the memory object. The name PDB is used for memory objects that represent program descriptor blocks. These blocks contain execution information about applications.

5.9.16 Display Local Heap Command

Syntax

dh

This command displays a list of the local memory objects in the local heap (if any) belonging to the current data segment. The command uses the current value of the **ds** register to locate the data segment and check for a local heap. The list of memory objects has the following form:

offset: *size* { **BUSY** | **FREE** }

The *offset* specifies the address offset from the beginning of the data segment to the local memory object. The *size* specifies the size of the memory object in bytes. If **BUSY** is given, the object has been allocated and is currently in use. If **FREE** is given, the object is in the pool of free objects ready to be allocated by the application. A special memory object, **SENTINAL**, may also be displayed.

5.9.17 Dump Long Reals Command

Syntax

dl [[*range*]]

This command displays the hexadecimal and decimal values of the long (8-byte) floating-point numbers in the *range*. Each display line shows the address of the floating-point number, the hexadecimal values of the bytes in the number, and the decimal value of the number.

5.9.18 Dump Task Queue Command

Syntax

dq

This command displays a list containing information about the various task queues supported by the system.

5.9.19 Dump Short Reals Command

Syntax

ds [[*range*]]

This command displays the hexadecimal and decimal values of the short (4-byte) floating-point numbers in the *range*. Each display line shows the address of the floating-point number, the hexadecimal values of the bytes in the number, and the decimal value of the number.

5.9.20 Dump Ten-Byte Reals Command

Syntax

dt [[*range*]]

This command displays the hexadecimal and decimal values of the ten-byte floating-point numbers in the *range*. Each display line shows the address of the floating-point number, the hexadecimal values of the bytes in the number, and the decimal value of the number.

5.9.21 Dump Words Command

Syntax

dw [[*range*]]

This command displays the hexadecimal values of the words (2-byte values) in the *range*. Each display line shows the address of the first word in the line and up to eight hexadecimal word values.

5.9.22 Enter Command

Syntax

e *address* [[*list*]]

This command enters one or more values into memory. The size of the value entered depends on the most recently used enter command. (Enter commands are **e**, **ea**, **eb**, **ed**, **el**, **es**, **et**, and **ew**.) Bytes are the default. If no *list* is given, the command displays the value at *address* and prompts for a new value. If *list* is given, the command replaces the value at *address* and at each subsequent address until all values in the list have been used.

5.9.23 Enter Address Command

Syntax

ea *address* [[*list*]]

This command enters an ASCII string into memory. If no *list* is given, the command displays the byte at the given *address* and prompts for a replacement. If *list* is given, the command replaces the bytes at the given address, then displays the next byte and prompts for a replacement.

5.9.24 Enter Bytes Command

Syntax

eb *address* [[*list*]]

This command enters one or more byte values into memory. If *list* is given, the command replaces the byte at *address* and bytes at each subsequent address until all values in the list have been used. If no *list* is given, the command displays the byte at *address* and prompts for a new value. To skip to the next byte, enter a new value, or press the SPACEBAR. To move back to the previous byte, type a hyphen (–). To exit the command, press the ENTER key.

5.9.25 Enter Double-words Command

Syntax

ed *address* [[*value*]]

This command enters a double-word value into memory. If no *value* is given, the command displays the double-word at the given *address* and prompts for a replacement. If a *value* is given, the command replaces the double-word at the given address, then displays the next double-word and prompts for a replacement. Double-words must be typed as two words separated by a colon.

5.9.26 Enter Long Reals Command

Syntax

el *address* [[*value*]]

This command enters a long real value into memory. If no *value* is given, the command displays the long real value at the given *address* and prompts for a replacement. If *value* is given, the command replaces the long real value at the given address, then displays the next long real value and prompts for a replacement.

5.9.27 Enter Short Reals Command

Syntax

es *address* [[*value*]]

This command enters a short real value into memory. If no *value* is given, the command displays the short real value at the given *address* and prompts for a replacement. If *value* is given, the command replaces the short real value at the given address, then displays the next short real value and prompts for a replacement.

5.9.28 Enter Ten-Byte Reals Command

Syntax

et *address* [[*value*]]

This command enters a ten-byte real value into memory. If no *value* is given, the command displays the ten-byte real value at the given *address* and prompts for a replacement. If *value* is given, the command replaces the ten-byte real value at the given address, then displays the next ten-byte real value and prompts for a replacement.

5.9.29 Enter Words Command

Syntax

ew *address* [[*value*]]

This command enters a word value into memory. If no *value* is given, the command displays the word at the given *address* and prompts for a replacement. If *value* is given, the command replaces the word at the given address, then displays the next word and prompts for a replacement.

5.9.30 Fill Command

Syntax

f *range list*

This command fills the addresses in the given *range* with the values in the *list*. If *range* specifies more bytes than the number of values in the list, the list is repeated until all bytes in the range are filled. If *list* has more values than the number of bytes in the range, the command ignores any extra values.

5.9.31 Go Command

Syntax

g [[=*startaddress*]] [[*breakaddress*]]...

This command passes execution control to the program at the given *start-address*. Execution continues to the end of the program or until a *breakaddress* is encountered. The program also stops at any breakpoints set using the **bp** command. If no *startaddress* is given, the command passes execution to the address specified by the current values of the **cs** and **ip** registers. If a *breakaddress* is given, it must specify an instruction address (that is, the address must contain the first byte of an instruction). Up to ten break addresses, in any order, can be given at one time.

5.9.32 Hex Command

Syntax

h *value1 value2*

This command displays the sum and difference of two hexadecimal numbers, *value1* and *value2*.

5.9.33 Input Command

Syntax

i *value*

This command reads and displays one byte from the input port specified by *value*. The *value* can be any 16-bit port address.

5.9.34 Backtrace Stack Command

Syntax

k [[*value*]]

This command displays the current stack frame. Each line shows the name of a procedure, its arguments, and the address of the statement that called it. The command displays two 2-byte arguments by default. If *value* is given, the command displays that many 2-byte arguments. Using the **k** command at the beginning of a function (before the function prolog has been executed) may give incorrect results. The command uses the **bp** register to compute the current backtrace, and this register is not correctly set for a function until its prolog has been executed.

5.9.35 Backtrace Task Stack Command

Syntax

kt *pdb* [[*value*]]

This command displays the stack frame of the program specified by the *pdb*. Each line shows the name of a procedure, its arguments, and the address of the statement that called it. The command displays two 2-byte arguments by default. If *value* is given, the command displays that many 2-byte arguments. The *pdb* must be the segment address of the program descriptor block for the task to be traced.

5.9.36 Load Command

Syntax

l [[*address* [[*drive record count*]]]]

This command copies the contents of a named file or the contents of a given number of logical disk records into memory. The contents are copied to the given *address* or to a default address, and the **bx**:**cx** register pair is set to the number of bytes loaded.

To load a file, set the filename using the **n** command (otherwise, **symdeb** uses whatever name is currently at location **ds**:5**c**). If *address* is not given, **symdeb** copies bytes to **cs**:100.

To load logical records from a disk, set *drive* to zero (drive A), 1 (drive B), or 3 (drive C). Set *record* to the first logical record to be read (any 1- to 4-digit hexadecimal number). Set *count* to the number of records to read (any 1- to 4-digit hexadecimal number). If the named file has a *.exe* extension, **l** adjusts the load address to the address given in the *.exe* file header. The command strips any header information from a *.exe* file before loading. If the named file has a *.hex* extension, the **l** command adds that file's start address to *address* before loading the file.

5.9.37 Move Command

Syntax

m *range address*

This command moves the block of memory specified by *range* to the location starting at *address*. All moves are guaranteed to be performed without data loss.

5.9.38 Macro Command

Syntax

m[[*id*]][[=*command-string*]]

This command defines or executes a **symdeb** command macro. The *id* identifies the macro to be defined or executed. There are ten macros, numbered 0 through 9. If *command-string* is specified, the command assigns the **symdeb** commands given in the string to the macro. If no string is given, the command executes the commands currently assigned to the macro. Macros are initially empty unless the /@ option is used when **symdeb** is started. This option reads a set of macro definitions from a specified file.

5.9.39 Name Command

Syntax

n [[*filename*]] [[*arguments*]]

This command sets the filename for subsequent **l** and **w** commands, or sets program arguments for subsequent execution of a loaded program. If *filename* is given, all subsequent **l** and **w** commands will use this name when accessing disk files. If *arguments* are given, the command copies all arguments, including spaces, to the memory location starting at **ds**:81 and sets the byte at **ds**:80 to a count of the total number of characters copied. If the first two *arguments* are also filenames, the command creates file control blocks at addresses **ds**:5C and **ds**:6C and copies the names (in proper format) to these blocks.

5.9.40 Output Command

Syntax

o *value byte*

This command sends the given *byte* to the output port specified by *value*. The *value* can be any 16-bit port address.

5.9.41 Program Step Command

Syntax

p [[=*startaddress*]] [[*value*]]

This command executes the instruction at the given *startaddress*, then displays the current values of all registers and flags. If *startaddress* is given, the command starts execution at the given address. Otherwise, it starts execution at the instruction pointed to by the current **cs** and **ip** registers. If *value* is given, the command executes *value* number of instructions before stopping. The command automatically executes and returns from any call instructions or software interrupts it encounters, leaving execution control at the next instruction after the call or interrupt.

5.9.42 Quit Command

Syntax

q

This command terminates **symdeb** execution and returns control to DOS.

5.9.43 Register Command

Syntax

r [[*register*[[=]*value*]]]

This command displays the contents of CPU registers and allows the contents to be changed to new values.

If no *register* is specified, the command displays all registers, flags, and the instruction at the address pointed to by the current **cs** and **ip** register values. If *register* is specified, the command displays the current value of the register and prompts for a new value. If both *register* and *value* are specified, the command changes the register to the specified value.

5.9.44 Search Command

Syntax

s *range list*

This command searches the given *range* of memory locations for the byte values given in *list*. The command displays the address of each byte found.

5.9.45 Set Source Mode Commands

Syntax

s–
s&
s+

These commands set the display mode for commands that display instruction code. If **s–** is given, **symdeb** disassembles and displays the instruction code in memory. If **s&** is given, **symdeb** displays both the actual program source line and the disassembled code. If **s+** is given, **symdeb** displays the actual program source line corresponding to the instruction to be displayed.

To access a source file for the first time, **symdeb** may display the following prompt:

```
Source file name for mapname (cr for none)?
```

In such cases, type the name, including extension, of the source file corresponding to the symbol file *mapname*.

5.9.46 Trace Command

Syntax

t [[=*startaddress*]] [[*value*]]

This command executes the instruction at the given *startaddress*, then displays the current values of all registers and flags. If *startaddress* is given, the command starts execution at the given address. Otherwise, it starts execution at the instruction pointed to by the current **cs** and **ip** registers. If *value* is given, the command continues to execute *value* number of instructions before stopping. In source-only mode (**s+**), **t** operates directly on source lines. The **t** command can be used to trace instructions in ROM.

5.9.47 Unassemble Command

Syntax

u [[*range*]]

This command displays the instructions and/or statements of the program being debugged. The **s** command sets the display format. If *range* is given, the command displays instructions generated from code within the given range. Otherwise, the command displays the instructions generated from the first eight lines of code at the current address. 80286 protected-mode mnemonics cannot be displayed.

5.9.48 View Command

Syntax

v *range*

This command displays source lines beginning at the specified address. The symbol file must contain line-number information.

5.9.49 Write Command

Syntax

w [[*address* [[*drive record count*]]]]

This command writes the contents of a given memory location to a named file, or to a given logical record on disk. To write to a file, set the filename with an **n** command, and set the **bx**:**cx** register pair to the number of bytes to be written. If no *address* is given, the command copies bytes starting from the address **cs**:100, where **cs** is the current value of the **cs** register. To write to a logical record on disk, set the *drive* to any number in the range zero (drive A) to 3 (drive C), set the *record* to the first logical record to receive the data (a 1- to 4-digit hexadecimal number), and set the *count* to the number of records to write to the disk (a 1- to 4-digit hexadecimal number). Do not write data to an absolute disk sector unless you are sure the sector is free.

5.9.50 Examine Symbol Map

Syntax

x [[?]] *symbol*

This command displays the name and load-segment addresses of the current symbol map, segments in that map, and symbols within those segments.

If **?** is not given, the command displays the current symbol map name and the segments within that map. If the asterisk (*) is specified, the command displays the names and load-segment addresses for all currently loaded symbol maps.

If **?** is given, the command displays all symbols within the given symbol map that match the *symbol* specification. A *symbol* specification has the following form:

[[*mapname*!]] [[*segmentname*:]] [[*symbolname*]]

If *mapname*! is given, the command displays information for that symbol map. The *mapname* must be the filename (without extension) of the corresponding symbol file.

If *segmentname*: is given, the command displays the name and load-segment address for that segment. The *segmentname* must be the name of a segment named within the explicitly given or currently open symbol map.

If *symbolname* is given, the command displays the segment address and segment offset for that symbol. The *symbolname* must be the name of a symbol in the given segment.

To display information about more than one segment or symbol, enter a partial *segmentname* or *symbolname* ending with an asterisk (*). The asterisk acts as a wildcard character.

5.9.51 Open Symbol Map Command

Syntax

xo [[*symbol*!]]

This command sets the active symbol map and/or segment. If *symbol*! is given, the command sets the active symbol map to the given map. The *symbol* must be the filename (without extension) of one of the symbol files specified in the **symdeb** command line. A map file can be opened only if it was loaded by providing its name in the **symdeb** command line.

5.9.52 Set Symbol Value Command

Syntax

z *symbol value*

This command sets the address of *symbol* to *value.*

5.9.53 Display Help Command

Syntax

?

This command displays a list of all **symdeb** commands and operators.

5.9.54 Display Expression Command

Syntax

? *expression*

This command displays the value of *expression.* The display includes a full address, a 16-bit hexadecimal value, a full 32-bit hexadecimal value, a decimal value (enclosed in parentheses), and a string value (enclosed in double quotation marks). The *expression* can be any combination of numbers, symbols, addresses, and operators.

5.9.55 Source Line Display Command

Syntax

.

This command displays the current source line.

5.9.56 Redirect Input Commands

Syntax

< filename
{ filename

The **<** command causes **symdeb** to read all subsequent command input from the given file. The { command reads all input for the debugged program from the given file.

5.9.57 Redirect Output Commands

Syntax

> filename
} filename

The **>** command causes **symdeb** to write all subsequent command output to the given file. The } command writes all output from the debugged program to the given file.

5.9.58 Redirect Input and Output Commands

Syntax

=filename
~ filename

The **=** command causes **symdeb** both to read and write to the given device. The ~ command causes the debugged program both to read from and write to the given device.

5.9.59 Shell Escape Command

Syntax

! [[*dos-command*]]

This command passes control to *command.com*, the DOS command processor, letting the user carry out DOS commands. The DOS **exit** command returns control to **symdeb**. If *command* is given, **symdeb** passes *dos-command* to *command.com* for execution, then receives control back as soon as the command is completed.

5.9.60 Comment Command

Syntax

comment

This command echos *comment* on the screen (or other output device).

Chapter 6

A Program Maintainer: Make

6.1 Introduction

The Microsoft Program Maintenance Utility (**make**) automates the process of maintaining assembly-language and high-level-language programs. The **make** utility automatically carries out all the tasks that need to be done in order to update a program after one or more of its source files have changed.

Unlike other batch-processing programs, **make** does not assemble, compile, and link all files just because one file has been updated. The **make** utility compares the last modification date of the file or files that may need updating with the modification dates of files on which these target files depend. The **make** utility then carries out the given task only if a target file is out of date. This process saves you time when you are creating programs that have many source files or that take several steps to complete.

This chapter explains how to use **make** and illustrates how to maintain a sample assembly-language program.

6.2 Using Make

To use **make**, you must create a **make** description file that defines the tasks you want to accomplish and specifies the files on which these tasks depend. Once the description file exists, you invoke **make** and supply the filename as a parameter; **make** then reads the contents of the file and carries out the requested tasks. Sections 6.2.1 and 6.2.2 explain how to create a **make** description file and how to start **make**.

6.2.1 Creating a Make Description File

You can create a **make** description file with a text editor. A **make** description file consists of one or more target and/or dependent descriptions. Each description has the following general form:

targetfile : *dependentfiles*
    command1
    [[*command2*]]
    .
    .
    .

The *targetfile* is the name of a file that may need updating. The *dependentfile* is the name of a file on which the target file depends.

The *targetfile* and *dependentfile* must be valid filenames. A pathname must be provided for any file that is not on the same drive and directory as the description file.

Any number of dependent files can be given, but only one target name is allowed. The names of dependent files must be separated from each other by at least one space. If you have more dependent filenames than can fit on one line, you can continue the names on the next line by typing a backslash (\) followed by a new line.

The *command* can be any valid DOS command line consisting of the name of an executable filename or a DOS internal command. Any number of commands can be given, but each must begin on a new line and must be preceded by a tab or by at least one space. The commands are carried out only if one or more of the dependent files has been modified since the target file was created.

Think of the **make** format as an "if/then" statement: If any *dependentfile* is newer than the *targetfile*, or if there is no *targetfile*, then execute *commands*.

You can give any number of target or dependent descriptions in a description file. You must make sure, however, that the last line in one description is separated from the first line of the next by at least one blank line.

The pound character (#) is a comment character. All characters after the comment character on the same line are ignored. When comments appear in a command-line section, the comment character (#) must be the first character on the line (no white space should appear before it). On any other lines, the comment character can appear anywhere.

Note

The order in which you place the target or dependent descriptions is important. The **make** utility examines each description in turn and bases its decision to carry out a given task on the file's current modification date. If a command in a later description modifies a file, **make** cannot return to the description in which that file is a target.

Example

```
startup.obj:        startup.asm
        masm startup,startup,nul,nul

print.obj:          print.asm
        masm print,print,print,print

print.ref:          print.crf
        cref print,print

print.exe:          startup.obj print.obj \lib\syscal.lib
        link startup+print,print,print/map,\lib\syscal;

print.sym:          print.map           #make a symbol file for debugging
#use the -l option to print information
        mapsym -l print.map
```

This example defines the actions needed to create five target files. Each file has at least one dependent file and one command. The target descriptions are given in the order in which the target files will be created. Thus, *startup.obj* and *print.obj* are examined and created, if necessary, before *print.exe*.

Notice that a comment appears on the same line as the target description for *print.sym*. However, in the command-line section, the comment appears on a separate line because the comment character (#) must be the first character on the line.

6.2.2 Starting Make

The **make** command line has the following form:

make [[*options*]] [[*macrodefinitions*]] *filename*

The *options* are one or more of the options described in Section 6.2.3. The *macrodefinitions* are one or more of the macro definitions described in Section 6.2.4. The *filename* is the name of a **make** description file. A **make** description file, by convention, has the same filename (but with no extension) as the program it describes. Although any filename can be used, this convention is preferred.

Once you start **make**, it examines each target description in turn. If a given target file is out-of-date compared to its dependent file or if the target file does not exist, **make** executes the given command or commands. Otherwise, it skips to the next target description.

When **make** finds an out-of-date target file, it displays the command or commands from the target/dependent description, then executes the commands. If **make** cannot find a specified file, it displays a message informing you that the file was not found. If the missing file is a target file, **make** continues execution because the missing file will, in many cases, be created by subsequent commands.

If the missing file is a dependent or command file, **make** stops execution of the description file; **make** also stops execution and displays the exit code if the command returns an error. (You can override this by using the **/i** option. See Section 6.2.3 for details.)

When **make** executes a command, it uses the same environment used to invoke **make**. Thus, environment variables such as PATH are available for these commands.

6.2.3 Using Make Options

The options available with the **make** command modify its behavior and are described as follows:

Option	Action
/d	This option causes **make** to display the last modification date of each file as the file is scanned.
/i	This option causes **make** to ignore exit codes (also called return or "errorlevel" codes) returned by programs called by the **make** description file. Despite the errors, **make** will continue execution of the next lines of the description file.

/n	This option causes **make** to display commands that would be executed by a description file, but the commands are not actually executed.
/s	This option causes **make** to execute in "silent" mode. That is, lines are not displayed as they are executed.

Examples

```
make /n test
```

The first example directs **make** to display commands from the **make** description file named *test* without executing them.

```
make /d test
```

The second example directs **make** to execute the instructions from *test*, displaying the last modification date of each file as it is scanned.

6.2.4 Using Macro Definitions

Macro definitions allow you to associate a symbolic name with a particular value. By using macro definitions, you can change values in the description file without having to edit every line that contains a particular value.

A macro definition has the following form:

name=value

The form for using a previously defined macro definition is as follows:

$(*name*)

Occurrences of the pattern $(*name*) in the description file are replaced with the specified *value*. The *name* is converted to uppercase; "flags" and "FLAGS" are equivalent. If you define a macro name but leave the *value* blank, the *value* will be a null string.

Macro definitions can be placed in the **make** description file or given on the **make** command line. A *name* also is considered defined if it has a definition in the current environment. For example, if the environment variable PATH is defined in the current environment, occurrences of "$(PATH)" in the description file will be replaced with the PATH value.

In the **make** description file, each macro definition must appear on a separate line. Any white space (tab and space characters) between *name* and the equal sign (=) or between the equal sign and *value* is ignored. Any other white space is considered part of *value*. To include white space in a macro definition on the command line, enclose the entire definition in double quotation marks (" ").

If the same name is defined in more than one place, the following order of precedence applies:

1. Command-line definition
2. Description-file definition
3. Environment definition

Example

```
BASE=abc
BUF=/B63

$(BASE).obj:        $(BASE).asm
        masm $(BASE)  $(BUF),$(BASE),$(BASE),$(BASE)

$(BASE).exe:        $(BASE).obj \lib\math.lib
        link $(BASE),$(BASE),$(BASE)  /map,\lib\math
```

The preceding example of a **make** description file shows macro definitions for the names "BASE" and "BUF". The **make** utility replaces each occurrence of "$(BASE)" with "abc".

If the description file is called *assemble*, you can give the following command:

```
make BASE=def assemble
```

This command line enables you to override the definition of "BASE" in the description file, causing "def" to be assembled and linked instead of "abc".

If you want to override the 63K buffer size specified by the macro "BUF" in the **make** description file and instead use the **masm** default buffer size of 32K, you could start **make** with the following command line:

```
make BUF=assemble
```

Since the value for "BUF" is blank, it will be treated as a null string.

However, since the null string was given from the command line, which has higher precedence than the definition in the description file, "BUF" will be expanded to a null string and no option will be passed in the **masm** command line.

6.2.5 Nesting Macro Definitions

Macro definitions can be nested. In other words, a macro definition can include another macro definition. For example, you might have the following macro definition in the **make** description file *picture*:

```
LIBS=$(DLIB)\math.lib $(DLIB)\graphics.lib
```

You then could start **make** with the following command line:

```
make DLIB=d:\lib
```

In this case, every occurrence of the macro "LIBS" would be expanded to:

```
d:\lib\math.lib d:\lib\graphics.lib
```

Be careful to avoid infinitely recursive macros such as the following:

```
A = $(B)
B = $(C)
C = $(A)
```

6.2.6 Using Special Macros

The **make** utility recognizes three special macro names and will automatically substitute a value for each. The special names and their values are described as follows:

Name	Value Substituted
$*	Base-name portion of the target (without the extension)
$@	Complete target name
$**	Complete list of dependencies

These macro names can be used in description files, as shown in the following example:

```
test.exe: mod1.obj mod2.obj mod3.obj
          link $**, $@;
          mapsym $*
```

The preceding example is equivalent to the following:

```
test:exe: mod1.obj mod2.obj mod3.obj
          link mod1.obj mod2.obj mod3.obj, test.exe;
          mapsym test
```

6.2.7 Inference Rules

The **make** utility allows you to create inference rules that specify commands for target/dependent descriptions even when there is no explicit command in the **make** description file. An inference rule tells **make** how to produce a file with one type of extension from a file with the same base name and another type of extension. For example, if you define a rule for producing *.obj* files from *.asm* files, the actual commands do not have to be repeated in the description file for each target/dependent description.

Inference rules take the following form:

.dependentextension.targetextension :
    command1
    [[*command2*]]
    .
    .
    .

For lines that do not have explicit commands, **make** looks for a rule that matches both the target's extension and the dependent's extension. If it finds such a rule, **make** performs the commands given by the rule.

The **make** utility looks first for dependency rules in the current description file, but if it does not find an appropriate rule, it will search for the tools-initialization file, *tools.ini*, in the current drive and directory (or in any directories specified with the DOS **path** command).

If **make** finds *tools.ini*, it looks through the file for a line beginning with the tag "[make]". Inference rules following this line will be applied if appropriate.

Example

```
.obj.asm:
        masm $*.asm,,;

test1.obj: test1.asm

test2.obj: test2.asm
        masm test2.asm;
```

In the preceding sample description file, an inference rule is defined in the first line. The filename in the rule is specified with the macro name $* so that the rule will apply to any base name. When **make** encounters the dependency for files *test1.obj* and *test1.asm*, it looks first for commands on the next line. When it does not find any, **make** checks for a rule that may apply and finds the rule defined in the first lines of the description file. Then **make** applies the rule, replacing the $* macro with *test1* when it executes the following command:

```
masm test1.asm,,;
```

When **make** reaches the second dependency for the *test2* files, it does not search for a dependency rule because a command is explicitly stated for this target/dependent description.

6.3 Maintaining a Program: An Example

The **make** utility is especially useful for programs in development because it offers a quick way to re-create a modified program after small changes.

Suppose you have a test program named *test.asm* that you use to debug the routines in a library file named *math.lib*. The purpose of *test.asm* is to call one or more routines in the library so that a study of their interaction can be made. Each time *test.asm* is modified, it has to be assembled, a cross-reference listing has to be created, the assembled file has to be linked to the library, and finally a symbol file has to be created for use with the Microsoft Symbolic Debug Utility (**symdeb**).

These tasks will be carried out when the following target/dependent descriptions are copied to the **make** description file *test*:

```
test.obj:       test.asm
        masm test,test,test,test

test.ref:       test.crf
        cref test,test

test.exe:       test.obj \lib\math.lib
        link test,test,test/map,\lib\math

test.sym:       test.map
        mapsym /L test.map
```

These lines define the actions to be carried out to create four target files: *test.obj*, *test.ref*, *test.exe*, and *test.sym*. Each file has at least one dependent file and one command. The target/dependent descriptions are given in the order in which the target files will be created. Thus, *test.sym* depends on *test.map*, which is created by **link**; *test.exe* depends on *test.obj*, which is created by **masm**; and *test.ref* depends on *test.crf*, which also is created by **masm**.

Once the description file is in place, you can create *test.asm* using a text editor, then invoke **make** to create all other required files. The command line should have the following form:

```
make test
```

The **make** utility carries out the following steps:

1. It compares the modification date of *test.asm* with *test.obj*. If *test.obj* is out-of-date (or does not exist), **make** executes the following command:

   ```
   masm test,test,test,test
   ```

   Otherwise, it skips to the next target description.

2. It compares the dates of *test.ref* and *test.crf*. If *test.ref* is out-of-date, **make** executes the following command:

   ```
   cref test,test
   ```

3. It compares *test.exe* with the dates of *test.obj* and the library file *math.lib*. If *test.exe* is out-of-date with respect to either file, **make** executes the following command:

   ```
   link test,test,test/map,\lib\math.lib
   ```

4. It compares the dates of *test.sym* and *test.map*. If *test.sym* is out-of-date, **make** executes the following command:

```
mapsym /l test.map
```

When *test.asm* is first created, **make** will execute all commands because none of the target files exist. If you invoke **make** again without changing any of the dependent files, it will skip all commands. If you change the library file *math.lib* but make no other changes, **make** will execute the **link** command because *test.exe* is now out-of-date with respect to *math.lib*. It also will execute **mapsym** because *test.map* is created by **link**.

Windows Development Applications

Chapter 7
Icon Editor

7.1 Introduction

You can create customized icons, cursors, and bitmaps for your applications by using the Microsoft Windows Icon Editor. The Icon Editor lets you work on a large-scale icon, cursor, or bitmap while it displays the full-scale replica of your work.

Every application needs an icon to show that it is present, even if its window is not open. Every application needs a cursor to show that the mouse is active when moved into the application's window. Although Windows provides predefined icons and cursors, the Icon Editor lets you create your own unique icons and cursors for your applications.

The following sections explain how to use the Icon Editor.

Note

The Icon Editor must be used with a mouse or similar pointing device.

7.2 Starting the Icon Editor

The Icon Editor is a Windows application. To start it, open the MS-DOS Executive window and double-click the filename *iconedit.exe*. Windows loads the Icon Editor.

Figure 7.1 shows the window of the Icon Editor.

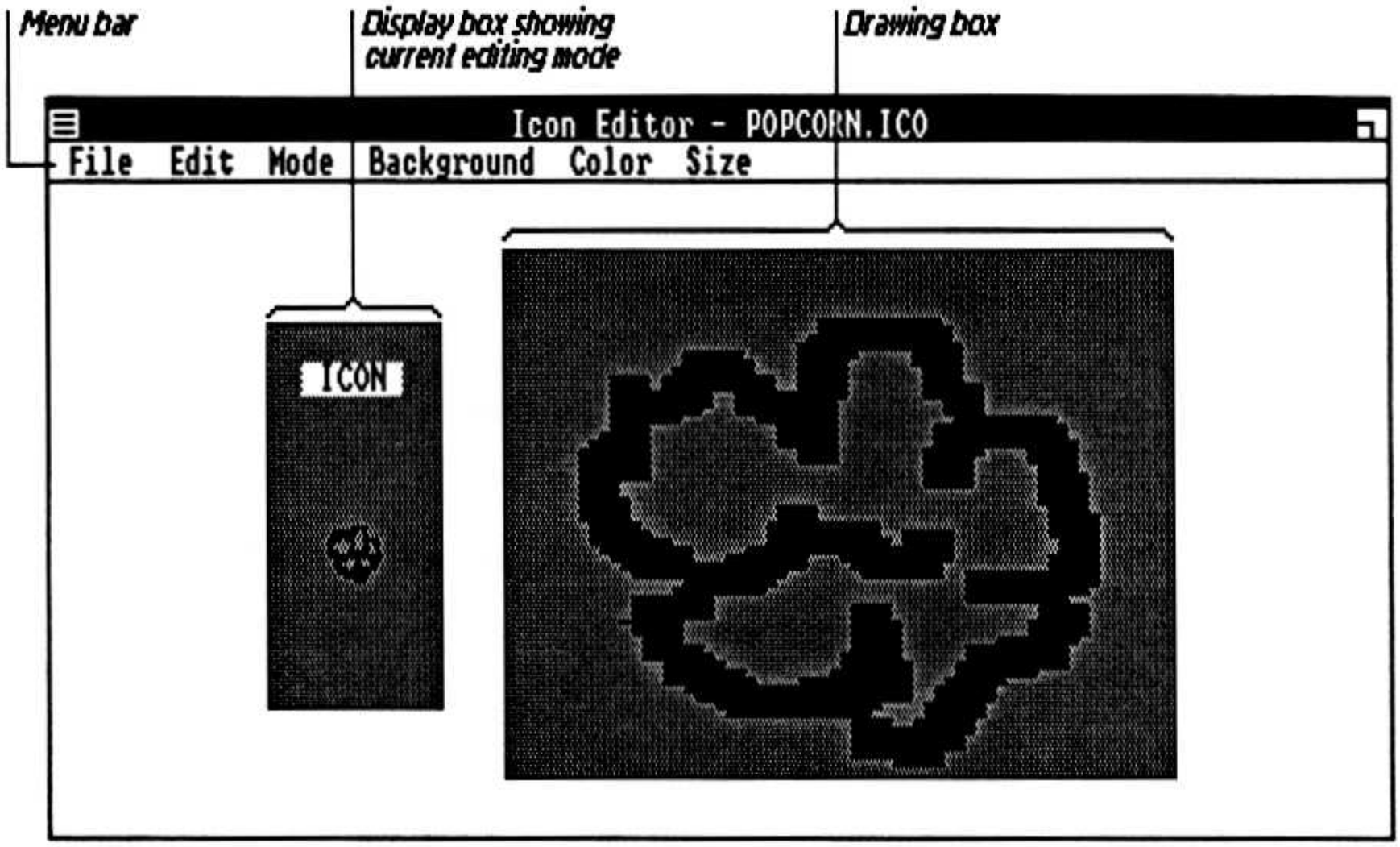

Figure 7.1 Icon Editor Window

The Icon Editor window has three main parts: the menu bar, which contains menu names; the display box; and the drawing box.

The menu bar lists the menu names at the top of the window. You can select a menu by pointing to the name with the mouse cursor and clicking the left mouse button. The Icon Editor has the following menus:

File
Edit
Mode
Background
Color
Size

The display box is the narrow box at the left of the screen. It contains the name of the current editing mode and a full-scale replica of your work. Initially, the editing mode is Icon.

The drawing box is the large, square box at the right of the screen. This box is the workspace for drawing your icons, cursors, and bitmaps. The box is a magnified view of a small part of the screen. Each pixel in this box is many times larger than those on the actual screen so that you can see

the individual pixels while doing your work. The box has an optional grid that you can use to help align pixels as you draw them.

7.3 Drawing in the Drawing Box

To draw an icon or cursor, you simply move the mouse cursor into the drawing box and use the left and right mouse buttons to color and erase pixels:

- To color a pixel, point to the pixel and click the left mouse button. The pixel takes on the current pen color.
- To color several pixels, use the left mouse button to drag the cursor over each pixel you want to color. The pixels take on the current pen color.
- To erase a pixel, point to the pixel and click the right mouse button. The pixel takes on the current background color.
- To erase several pixels, use the right mouse button to drag the cursor over each pixel you want to erase. The pixels take on the current background color.

Initially, the pen color is black and the background color is gray. You can change this by using the Color and Background commands described in Sections 7.6 and 7.9, respectively.

You can draw straight lines by holding down the SHIFT key while pressing a mouse button.

Occasionally, the lines you draw in the drawing box will not be reflected in the display box. You can remedy this by using the Redraw command. Choose Redraw from the Edit menu and the display box will be redrawn to correctly display the icon, cursor, or bitmap you have drawn.

7.4 Clearing the Drawing Box

You can remove the current contents of the drawing box (and the display box) by using the New command.

To clear the drawing box, choose the New command from the File menu. A new, empty file is opened and the drawing box and display box are cleared.

If you made changes to the current file that you did not save, you will see a dialog box asking about saving your changes. If you want to save the changes, choose Yes; otherwise, choose No. In either case, a new file is opened.

7.5 Choosing the Editing Mode

You can choose the editing mode for the Icon Editor by using the Mode menu. The Icon Editor has three modes: Icon, for editing icons; Cursor, for editing cursors; and Bitmap, for editing patterns.

To choose the Icon or Cursor mode, choose the Icon or Cursor command from the Mode menu. The Icon Editor will change the label in the display box to the new mode. If appropriate, it also will clear the drawing box and re-display the drawing grid.

In Icon mode, the drawing box is a 64×64 block of pixels, but in Cursor mode, the box is a 32×32 block of pixels. This means you will see a difference in the size of the drawing box when using the different modes.

To choose the Bitmap mode, follow these steps:

1. Choose Bitmap from the Mode menu. You will see a dialog box like the one shown in Figure 7.2.

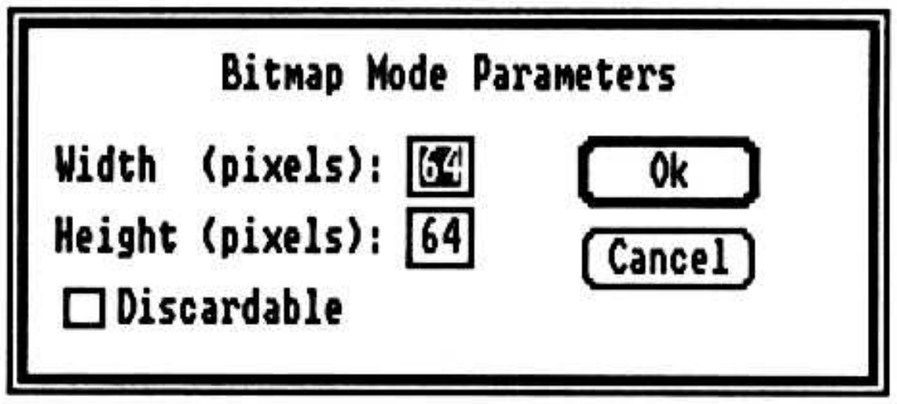

Figure 7.2 Bitmap Mode Parameters Dialog Box

2. If you want to change the width or height of the drawing box, select the appropriate box and type the new number. In Bitmap mode, the drawing box can be any size from 1×1 to 64×64. If you do not give a size, the default size is 64×64.
3. If you want the bitmap code to be a discardable resource, turn on the Discardable checkbox. Discardable resources help reduce memory usage by applications that do not require the use of the resource at all times.
4. Choose the Ok button or press the ENTER key.

The Icon Editor changes the label in the display box to Bitmap. If appropriate, it also will clear the drawing box and re-display the drawing grid. In Bitmap mode, the background color is set to black and cannot be changed. The pen color is set to white and can be changed only to black.

Notice that the Mode menu also contains the Hotspot command. This command sets the icon's or cursor's hotspot, described in Section 7.8.

7.6 Changing the Pen Color

You can change the color of the pen in the Icon Editor by using the Color menu. The current pen color defines what color the Icon Editor gives an icon or cursor pixel when you color it using the left mouse button.

The Color menu lists four pen colors: White, Black, Screen, and –Screen. An icon or cursor pixel with Screen color has the same color as the screen pixel underneath it. –Screen is the inverted color. An icon or cursor pixel with –Screen color has the opposite color of the screen pixel underneath it. For example, if the screen is white, the pixel is black.

For icons and cursors, you can use all four colors. In Bitmap mode, the pen can be only black or white.

To change the pen color, choose the desired color from the Color menu. The Icon Editor now sets the pen color to the desired color.

7.7 Changing the Pen Size

You can change the size of the pen by using the Size menu. The pen size determines the number of pixels you can color at once. The Size menu lists three pen sizes: Small, Medium, and Large.

To change the pen size, choose the size you want from the Size menu. The Icon Editor now sets the pen size to the desired size.

7.8 Setting the Hotspot

In an icon, the hotspot is the pixel that Windows uses to determine where (column and row) to place a window that you have dragged into the work area. In a cursor, the hotspot is the pixel from which Windows will take the cursor's current screen coordinates. To set the hotspot in an icon or cursor, follow these steps:

1. Choose the Hotspot command from the Mode menu. A checkmark appears next to it. Information about the current location of the hotspot appears beneath the display box.
2. Move the mouse pointer to the location in the icon or cursor where you want to place the hotspot and click the mouse button. The pointer changes to a bull's eye. The row and column information now indicates the location where you placed the hotspot. If you do not set the hotspot, the Icon Editor places it by default in the center (row 16, column 16 for cursors; row 32, column 32 for icons.)

Only one hotspot per icon or cursor is allowed. You can change the hotspot by dragging the bull's eye pointer to a new location in the cursor or icon.

If you want to continue drawing the icon or cursor after you have set the hotspot, you must again choose the Hotspot command from the Mode menu. When you do so, the checkmark next to Hotspot is removed and the row and column information disappears.

7.9 Changing the Background Color

You can change the background color of the drawing box and the display box by using the Background menu. The Background menu lists four background colors: Black, White, Gray, and Light Gray. To change the background color, choose the desired color command from the Background menu. The Icon Editor re-displays the drawing box and display box with the new background.

It is important to understand that the background color is used only for viewing. Changing the background color does not change what is stored in the icon, cursor, or bitmap file. Thus, you use the background color to see how an icon or cursor will look on a different background. For example, some icons may not look good on a gray background. By using the background colors, you can test this before adding the icon to your application.

When you open a new icon or cursor file, all the pixels in the file are Screen pixels. You create an icon or cursor by drawing pixels in the pen color. For icons and cursors, you have a choice of four pen colors: White, Black, Screen, and –Screen. To see how these look on different backgrounds, you can choose any of the four background colors.

Bitmaps, however, can have only white or black pixels. When you open a new bitmap, all the pixels are black. Since there cannot be any Screen or –Screen pixels in bitmaps, the background color does not matter. You will notice that you cannot choose different background colors for bitmaps.

For example, if you want to draw the outline of a black box on a white background, you must first paint the whole drawing box white, then use a black pen to draw the figure. Using the Background menu and choosing White will not save white pixels in your icon or cursor file.

7.10 Displaying the Drawing Grid

The drawing grid is a grid of lines displayed in the drawing box to help you locate the individual pixels of an icon or cursor.

To display the grid, choose the Grid command from the Edit menu. To remove the grid, choose Grid again.

7.11 Opening an Existing Icon, Cursor, or Bitmap File

You can open an existing icon, cursor, or bitmap file using the Open command in the File menu. Follow these steps:

1. Choose the Open command from the File menu. You will see the dialog box shown in Figure 7.3.

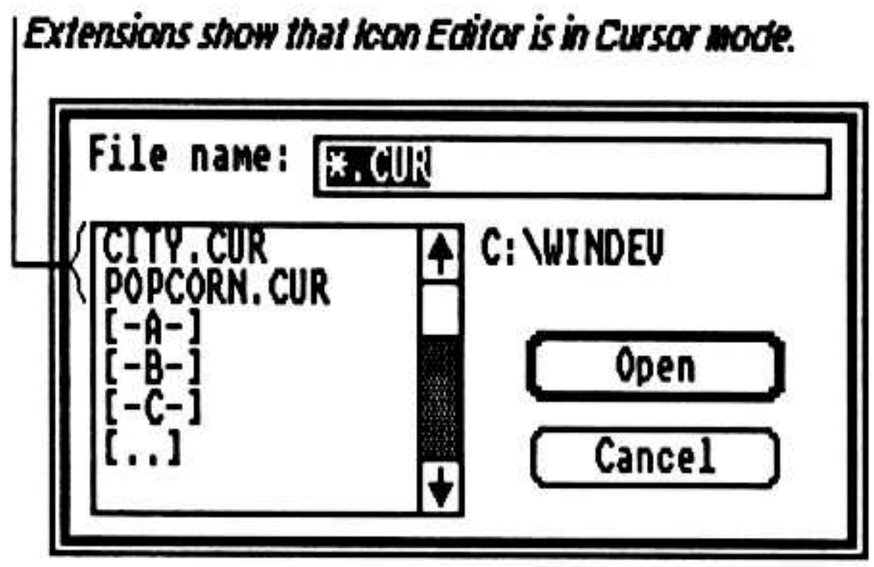

Figure 7.3 Open File Dialog Box

   The list box displays files with the extension *.ico*, *.cur*, or *.bmp*, depending on whether the Icon Editor is currently in Icon, Cursor, or Bitmap mode.

2. Open the file using any of the following methods:

   - Double-click the filename in the list box.

- Select the filename in the list box and choose the Open command or press the ENTER key.
- Type the filename in the text box and choose the Open command or press the ENTER key.

If you decide not to open a file, choose the Cancel button.

The Icon Editor opens the given file. It automatically changes the editing mode, if necessary. This means you can open icon, cursor, or bitmap files in any mode. If you have made changes to a file but haven't saved them, the Icon Editor creates a dialog box that asks if you want to save the changes.

7.12 Opening a New File

You can open a new icon, cursor, or bitmap file by choosing the New command from the File menu. When you do so, the display box and drawing box are cleared and you can begin drawing a new figure. The mode stays the same until you change it. If you were working on an icon, cursor, or bitmap and did not save the changes, a dialog box prompts you to save them. (For more information on saving files, see Section 7.13.)

7.13 Saving Files

You can save an icon, cursor, or bitmap in a file by using the Save or Save As command in the File menu. To save a file under a current filename, choose the Save command from the File menu. (If you attempt to save a file that doesn't have a filename, you will see the Save As dialog box, described in the following paragraphs.)

To save a file if there is no current filename, or if you want to change the filename, follow these steps:

1. Choose Save As from the File menu. You will see the dialog box shown in Figure 7.4.

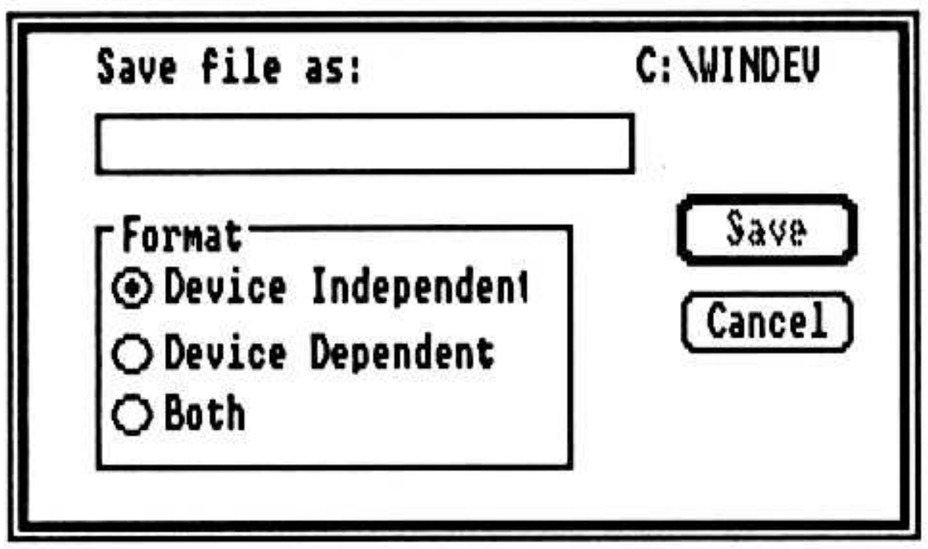

Figure 7.4 Save As Dialog Box

2. Type a valid filename in the text box. If you do not type an extension, the Icon Editor automatically supplies an extension based on the current editing mode: *.ico* for Icon mode, *.cur* for Cursor mode, and *.bmp* for Bitmap mode.

3. Select the appropriate radio button: Device Independent, Device Dependent, or Both. If you create a device-dependent file, the cursor, icon, or bitmap will always look best on the device on which it was created and may not look like you intended it to on other devices. A device-independent item will look acceptable on any device. In general, you should save all icons, cursors, and bitmaps as device-independent files.

4. Choose the Save button, or press the ENTER key.

The Icon Editor saves the icon, cursor, or bitmap in the named file.

7.14 Icon Display

The Icon Editor's 64×64 pixel display format for icons is based on a 1024×1024 pixel display screen. If your screen has fewer pixels, the Icon Editor will automatically adjust the full-scale image of your work (shown in the display box) to an image appropriate for your display-screen type.

For example, on 640×200 pixel monitors commonly used with the IBM PC, the 64×64 pixel icon is reduced to a 32×16 pixel block. The Icon Editor simply deletes every other column and three of every four rows. As a result, the icon that appears in the display box is not necessarily an exact replica of your work.

Chapter 8
Font Editor

8.1 Introduction

The Microsoft Windows Font Editor lets you create your own font files to use with your applications. A font file consists of a header and a collection of character bitmaps that represent the individual letters, digits, and punctuation characters that can be used to display text on a display surface. This chapter describes how to use the Windows Font Editor to create fonts.

Application writers who want to use fonts in their applications must add the new font files to a font resource file. For a description of how to make a font resource file, see the *Microsoft Windows Programmer's Reference.*

Note

The Font Editor can be used only to create and edit raster fonts. Vector font files are created as described in the *Microsoft Windows Programmer's Reference.*

The Font Editor must be used with a mouse or similar pointing device.

8.2 Starting the Font Editor

The Font Editor is a Windows application. To start it, open the MS-DOS Executive window and double-click the filename *fontedit.exe.* Windows loads the Font Editor.

When you start the Font Editor for the first time, it displays a dialog box, which lets you select the font file you want to edit. The dialog box is described in Section 8.3.

8.3 Opening a Font File

The Font Editor does not allow you to create fonts from scratch. Instead, you open an existing font file, then make changes to it. You can open a font file by using the Open command in the File menu. To do so, follow these steps:

1. Choose Open from the File menu.

   The Font Editor displays a dialog box that contains a directory listing and a text box for entering a filename.

2. Select the font file you want to open by using one of the following methods:

   - Select the text box, then use the keyboard to type the font filename. If the box already contains a name, you can delete it by typing a different filename. Choose the Open button or press the ENTER key.
   - Double-click a filename in the directory window. The selected file is opened.
   - If the file you want is in another directory or on another disk, select the directory or disk. These names are enclosed in brackets ([ ]).

Once a font file is opened, you will see the font's characters displayed in the font window and one of the characters displayed in the character window as shown in Figure 8.1.

Note

The Font Editor displays an error message if an attempt is made to load a file that does not contain a font.

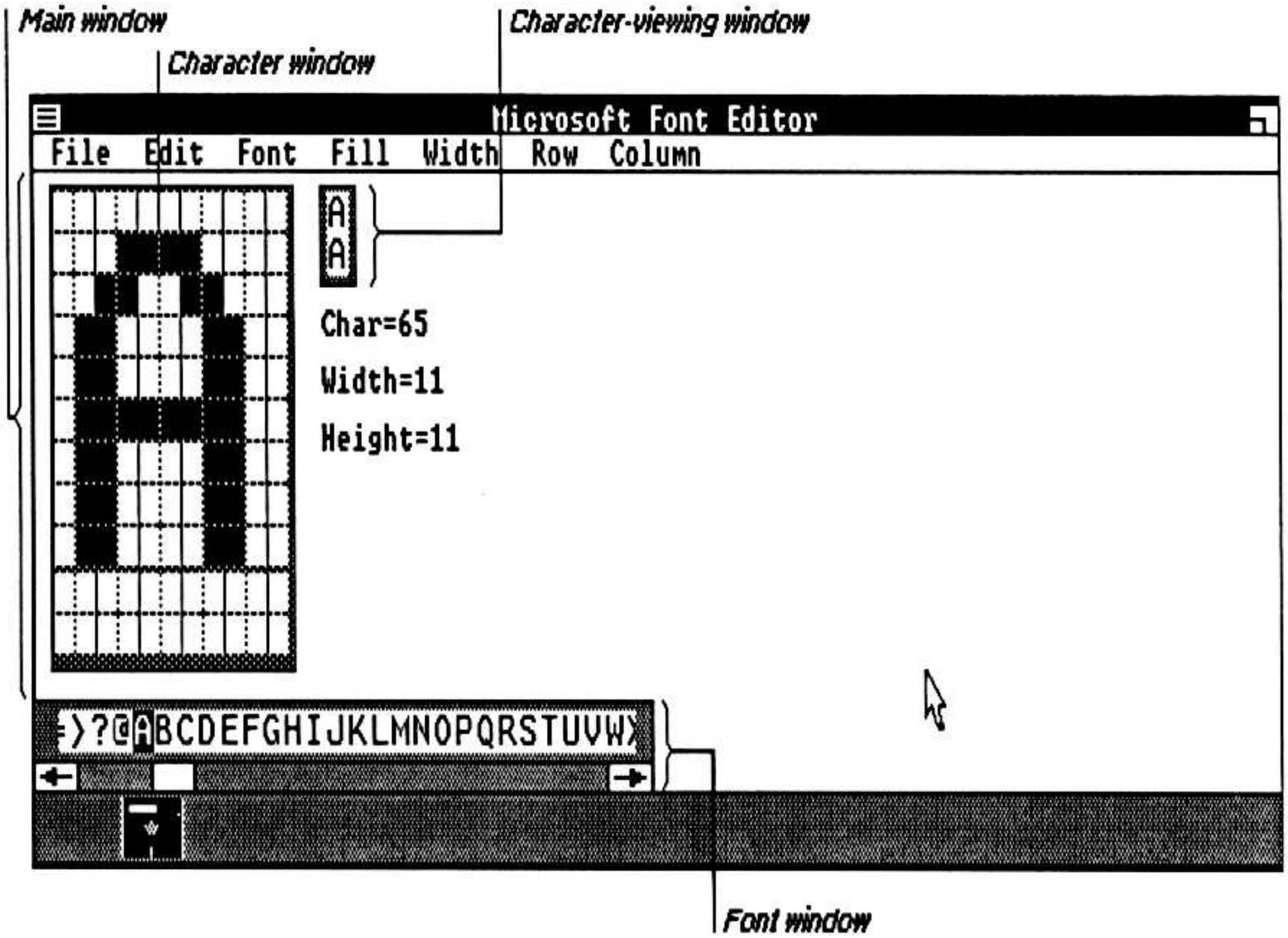

Figure 8.1 Font Editor Window

8.4 Font Editor Features

The Font Editor has the following features:

Feature	Description
Main window	This window contains the Font Editor's working windows. The menu bar contains the following menus: File, Edit, Font, Fill, Width, Row, and Column.
Character window	This window appears immediately below the menu bar and contains a copy of the character you want to edit. The window is divided by a grid into several rectangles. Each rectangle in the grid represents a single character pixel. You edit a character by turning these

	pixels on or off, or by adding or deleting pixels.
Character-viewing window	This small window appears to the right of the character window; it contains two full-scale copies of the character in the character window. The character-viewing window lets you examine the effect of the changes you make to the character. It also lets you see the character's leading (the amount of vertical separation between lines). Below the character-viewing window is a list of important information about the character, such as its ANSI value and its width and height in pixels.
Font window	This window appears at the bottom of the main window; it contains a font-viewing area and a scroll bar. The font-viewing area displays full-scale copies of the characters in the font. The scroll bar lets you scroll the font-viewing area whenever there are more characters in the font than can fit.

8.5 Selecting a Character to Edit

You can select and edit any character in the currently loaded font. To select a character, follow these steps:

1. Move the cursor into the font-viewing area of the font window.
2. Click the character you want to edit.

The Font Editor highlights your selection and copies it to the character window.

Important

When you make a new selection, the Font Editor saves the old selection by copying it back to the font. If you do not want to save the changes you've made to the old selection, make sure you cancel these changes, using the Refresh command in the Edit menu, before you make the new selection.

8.6 Changing Pixels in a Character

You can change the pixels in a character by turning them off if they're on, or on if they're off. The "on" pixels make up the character shape or face and appear in the current text foreground color. The "off" pixels make up the character background and appear in the current background color. Changing the pixels changes the shape of the character.

To turn a background pixel on:

- Point to the pixel and click the mouse button.

To turn a foreground pixel off:

- Point to the pixel and click the mouse button.

To turn several background pixels on:

- Point to a background pixel, then drag the cursor over the pixels you want to change.

To turn several foreground pixels off:

- Point to a foreground pixel, then drag the cursor over the pixels you want to change.

8.7 Canceling Changes to a Character

You can cancel all changes you have made to a character by using the Refresh command in the Edit menu. The command replaces the current character in the character window with a copy from the font window.

Warning

You cannot cancel changes to a character by selecting a new character. Selecting a new character, or even reselecting the current character, causes Windows to save all changes. Only the Refresh command cancels changes.

If you have made changes you do not want, do not save the character.

8.8 Changing a Character's Width

You can change the width of a character belonging to a variable-pitch font by using the Width menu. The Width menu changes the number of columns in the character's bitmap by letting you choose one of the following commands:

Command	Action
Wider (left)	Adds a blank column to the left side of the character.
Wider (right)	Adds a blank column to the right side of the character.
Wider (both)	Adds a blank column to each side of the character.
Narrower (left)	Deletes a column from the left side of the character.
Narrower (right)	Deletes a column from the right side of the character.
Narrower (both)	Deletes a column from each side of the character.

To change a character's width, choose the desired command from the

Width menu. The Font Editor changes the character window to show the new width.

Note

The width of a character can be changed only on variable-pitch fonts.

Characters in a variable-pitch font cannot be wider than the maximum character width.

8.9 Copying a Row of Pixels

You can make a copy of a whole or partial row of pixels by using the Add command in the Row menu. This command inserts a new row between the selected row and the row immediately below it. The new row has the same pixels as the selected row.

To copy a row, follow these steps:

1. Choose Add from the Row menu.
2. Move the cursor to a pixel in the row you want to copy. To copy the entire row, point to the pixel at the far right. To copy a partial row, point to the right-most pixel you want to copy.
3. Click the mouse button.

When you copy a whole row, all rows below the selected row are pushed down one row, and the row at the bottom of the character window is pushed off the end. When you copy a partial row, only the selected pixel and the pixels to the left of the selected pixel are copied. The pixels in the rows below the copied pixels are pushed down as the new pixels are inserted, but the pixels to the right remain unchanged.

8.10 Deleting a Row of Pixels

You can delete a whole or partial row of pixels by using the Delete command in the Row menu. This command deletes a selected row and causes rows below it to move up one row.

To delete a row, follow these steps:

1. Choose Delete from the Row menu.
2. Move the cursor to a pixel in the row you want to delete. To delete the entire row, point to the pixel at the far right. To delete a partial row, point to the right-most pixel you want to delete.
3. Click the mouse button.

When you delete a whole row, all rows below the selected row move up one row, and a blank row appears at the bottom of the character window. When you delete a partial row, only the selected pixel and the pixels to the left of the selected pixel are deleted. The pixels in the rows below the deleted pixels move up, but the pixels to the right remain unchanged.

8.11 Copying a Column of Pixels

You can make a copy of a whole or partial column of pixels by using the Add command in the Column menu. This command inserts a new column between the selected column and the column immediately to the right. The new column has the same pixels as the selected column.

To copy a column, follow these steps:

1. Choose Add from the Column menu.
2. Move the cursor to a pixel in the column you want to copy. To copy the entire column, point to the pixel at the bottom. To copy a partial column, point to the lowest pixel you want to copy.
3. Click the mouse button.

When you copy a whole column, all columns to the right of the selected column are pushed right one column, and the column at the far right of the character window is pushed off the side. When you copy a partial column, only the selected pixel and the pixels above it are copied. The

pixels in the columns to the right of the copied pixels are pushed right as the new pixels are inserted, but the pixels below remain unchanged.

8.12 Deleting a Column of Pixels

You can delete a whole or partial column of pixels by using the Delete command in the Column menu. This command deletes a selected column and causes columns below it to move one row to the left.

To delete a column, follow these steps:

1. Choose Delete from the Column menu.
2. Move the cursor to a pixel in the column you want to delete. To delete the entire column, point to the pixel at the bottom. To delete a partial column, point to the lowest pixel you want to delete.
3. Click the mouse button.

When you delete a whole column, all columns to the right of the selected column move left one column, and a blank column appears at the right side of the character window. When you delete a partial column, only the selected pixel and the pixels above it are deleted. The pixels in the columns to the right of the deleted pixels move left, but the pixels below remain unchanged.

8.13 Clearing the Character Window

You can remove a block of foreground pixels from the character window by using the Clear command in the Fill menu. Follow these steps:

1. Choose Clear from the Fill menu.
2. Move the cursor to a pixel in the character window.
3. Press and hold the mouse button. This creates an anchor point (displayed in gray) that represents a corner of the block you want to clear.

4. Drag the cursor to another pixel.
5. Release the mouse button.

The Font Editor uses the anchor point and the second pixel as diagonally opposite corners of the block you want to clear. All pixels within the block are cleared. If you drag the cursor in a horizontal or vertical line, all pixels in the line are removed.

8.14 Filling the Character Window with a Solid Block

You can fill the character window with a solid block of foreground pixels by using the Solid command in the Fill menu. Follow these steps:

1. Choose Solid from the Fill menu.
2. Move the cursor to a pixel in the character window.
3. Press and hold the mouse button. This creates an anchor point (displayed in gray) that represents a corner of the block you want to fill.
4. Drag the cursor to another pixel.
5. Release the mouse button.

The Font Editor uses the anchor point and the second pixel as diagonally opposite corners of the block you want to fill. All pixels within the block become foreground pixels. If you drag the cursor in a horizontal or vertical line, all pixels in the line become foreground pixels.

8.15 Filling the Character Window with a Hatch

You can fill a block of the character window with a hatched pattern (alternate foreground and background) by using the Hatched command in the Fill menu. Follow these steps:

1. Choose Hatched from the Fill menu.
2. Move the cursor to a pixel in the character window.
3. Press and hold the mouse button. This creates an anchor point (displayed in gray) that represents a corner of the block you want to fill.
4. Drag the cursor to another pixel.
5. Release the mouse button.

The Font Editor uses the anchor point and the second pixel as diagonally opposite corners of the block you want to fill. Every other pixel within the block becomes a foreground pixel, all others become background. If you drag the cursor in a horizontal or vertical line, every other pixel in the line becomes a foreground pixel; all others become background pixels.

8.16 Inverting the Character Window

You can invert the pixels in a block in the character window by using the Inverted command in the Fill menu. (Foreground pixels become background, background pixels become foreground.) Follow these steps:

1. Choose Inverted from the Fill menu.
2. Move the cursor to a pixel in the character window.
3. Press and hold the mouse button. This creates an anchor point (displayed in gray) that represents a corner of the block you want to invert.
4. Drag the cursor to another pixel.
5. Release the mouse button.

The Font Editor uses the anchor point and the second pixel as diagonally opposite corners of the block you want to invert. All foreground pixels within the block become background pixels; all background pixels become foreground pixels.

8.17 Reversing the Character Window

You can reverse the pixels in a block in the character window by using the Left=Right or Top=Bottom command in the Fill menu. The Left=Right command reverses the block by "flipping" the contents of a block along a vertical line in the center of the block. The Top=Bottom command reverses the block by flipping the contents of a block along a horizontal line in the center of the block.

To reverse a block, follow these steps:

1. Choose either Left=Right or Top=Bottom from the Fill menu.
2. Move the cursor to a pixel in the character window.
3. Press and hold the mouse button. This creates an anchor point (displayed in gray) that represents a corner of the block you want to reverse.
4. Drag the cursor to another pixel.
5. Release the mouse button.

The Font Editor uses the anchor point and the second pixel as diagonally opposite corners of the block you want to reverse. If you drag the cursor in a horizontal or vertical line, the pixels in the line are reversed.

8.18 Copying or Pasting in the Character Window

You can copy a block of pixels to the Clipboard or fill a block with pixels from the Clipboard by using the Copy or Paste commands in the Fill menu. The Copy command copies the pixels in the block to the Clipboard. The Paste command fills the block with pixels from the Clipboard.

To copy or paste a block, follow these steps:

1. Choose Copy or Paste from the Fill menu.
2. Move the cursor to a pixel in the character window.

3. Press and hold the mouse button. This creates an anchor point (displayed in gray) that represents a corner of the block you want to copy or paste.
4. Drag the cursor to another pixel.
5. Release the mouse button.

The Font Editor uses the anchor point and the second pixel as diagonally opposite corners of the block you want to copy or paste.

Note

You can copy or paste the entire character window by using the Copy and Paste commands in the Edit menu.

8.19 Undoing a Change

You can recover from an editing mistake by using the Undo command in the Edit menu. This command restores the character window to what it was before the last change in the window.

To recover from a mistake, choose the Undo command from the Edit Menu. The Font Editor restores the character window to its state before the change.

The Undo command reverses changes made by the Fill, Row, Column, Width, Refresh, and Undo commands. It can also reverse changes made to individual pixels using the mouse.

The Undo command cannot undo changes made to a character that has been saved (that is, returned to the font).

8.20 Saving Changes to a Character

You can save the changes you have made to a character by following these steps:

1. Move the cursor into the font-viewing area of the font window.
2. Choose the character you are currently editing (it is highlighted) with the mouse button.

The Font Editor saves your selection by copying it back into the font. The font-viewing window is updated to show the new character.

Note

You can also save a character by making a new selection. The Font Editor saves the old selection before copying the new selection to the character window. This is useful if you want to continue editing characters in the same font.

8.21 Resizing the Font

You can change the height, width, and character mapping (ANSI value) of the font by using the Size command in the Font menu. The command displays a dialog box that contains the following options:

Option	Action
Character Pixel Height	Defines the height (in pixels) of the characters in the font.
Maximum Width (Variable-Width Fonts Only)	Defines the width (in pixels) of the widest character in the variable-pitch font.
Character Pixel Width (Fixed-Pitch Fonts Only)	Defines the width (in pixels) of all characters in the fixed-pitch font. In fixed-pitch fonts, all characters have equal width.

3. Press and hold the mouse button. This creates an anchor point (displayed in gray) that represents a corner of the block you want to copy or paste.
4. Drag the cursor to another pixel.
5. Release the mouse button.

The Font Editor uses the anchor point and the second pixel as diagonally opposite corners of the block you want to copy or paste.

Note

You can copy or paste the entire character window by using the Copy and Paste commands in the Edit menu.

8.19 Undoing a Change

You can recover from an editing mistake by using the Undo command in the Edit menu. This command restores the character window to what it was before the last change in the window.

To recover from a mistake, choose the Undo command from the Edit Menu. The Font Editor restores the character window to its state before the change.

The Undo command reverses changes made by the Fill, Row, Column, Width, Refresh, and Undo commands. It can also reverse changes made to individual pixels using the mouse.

The Undo command cannot undo changes made to a character that has been saved (that is, returned to the font).

8.20 Saving Changes to a Character

You can save the changes you have made to a character by following these steps:

1. Move the cursor into the font-viewing area of the font window.
2. Choose the character you are currently editing (it is highlighted) with the mouse button.

The Font Editor saves your selection by copying it back into the font. The font-viewing window is updated to show the new character.

Note

You can also save a character by making a new selection. The Font Editor saves the old selection before copying the new selection to the character window. This is useful if you want to continue editing characters in the same font.

8.21 Resizing the Font

You can change the height, width, and character mapping (ANSI value) of the font by using the Size command in the Font menu. The command displays a dialog box that contains the following options:

Option	Action
Character Pixel Height	Defines the height (in pixels) of the characters in the font.
Maximum Width (Variable-Width Fonts Only)	Defines the width (in pixels) of the widest character in the variable-pitch font.
Character Pixel Width (Fixed-Pitch Fonts Only)	Defines the width (in pixels) of all characters in the fixed-pitch font. In fixed-pitch fonts, all characters have equal width.

First Character	Defines the character value (for example, the ANSI value) of the first character in the font. The first character is the character to the far left when you scroll the contents of the font-viewing window all the way to the left.
Last Character	Defines the character value (for example, the ANSI value) of the last character in the font. The last character is the character to the far right when you scroll the contents of the font-viewing window all the way to the right.
Pitch	Defines the kind of font. Fixed and Variable are mutually exclusive. If Fixed is selected, the font is fixed-pitch. If Variable is selected, the font is variable-pitch.
Weight	Lists options that define the font weight, ranging from thin to heavy. Each option represents the specific degree of heaviness (i.e., thickness of stroke) of the font. The options are mutually exclusive.

Important

You can change a font from fixed-pitch to variable-pitch by selecting Variable in the Size dialog box. You cannot change a variable-pitch font to fixed-pitch.

To make a change to one of the height, width, first-character, or last-character options, follow these steps:

1. Choose the Size command in the Font menu. The Font Editor displays the Size dialog box, which contains the size options.
2. Select the number box of the option you want to change.
3. Use the BACKSPACE key to delete the existing number, then use the keyboard to type a new number.
4. Choose the Ok button or press the ENTER key.

The Font Editor immediately makes the changes you have requested. Once the changes are complete, the new font is displayed in the font-viewing window.

When you change the width or height of a font, the Font Editor stretches or compresses the existing characters to make new characters that have the given size. The resulting characters usually are very close to the desired appearance and can be corrected by hand, if necessary, to achieve the desired appearance.

8.22 Changing a Font File's Header Information

You can change the information in the font file's header by using the Header command in the Font menu. The Header command displays a dialog box that contains a listing of the information in the header. The list consists of the following items:

Item	Definition
Face Name	This character string is the name used to distinguish the font from other fonts. It is not necessarily the same as the font filename. It can be up to 32 characters.
File Name	This character string is the name of the font file being edited.
Copyright	This character string is either a copyright notice or additional information about the font. It can be up to 60 characters.
Nominal Point Size	This number defines the point size of the characters in the font. One point is equal to approximately 1/72 inch.
Height of Ascent	This number defines the distance (in pixels) from the top of an ascender to the baseline.
Nominal Vert. Resolution	This number defines the vertical resolution at which the characters were digitized.
Nominal Horiz. Resolution	This number defines the horizontal resolution at which the characters were digitized.
External Leading	This number defines the pixel height of the font's external leading. External leading is the number of rows at the bottom of a character

window (the rectangular region containing the character) that contain no foreground pixels.

Internal Leading

This number defines the pixel height of the font's internal leading. Internal leading is the number of rows at the top of a character window (the rectangular region containing the character) that contain no foreground pixels.

Default Character

This number defines the character value (for example, the ANSI value) of the default character. The default character is used whenever an attempt is made to display a character whose character value is less than that of the font's first character or greater than that of the font's last character.

Break Character

This number defines the character value of the break character. The break character is used to pad lines that have been justified. The break character is typically the space character. (For example, in the ANSI character set, the value is 32.)

ANSI or OEM

These options define the character set. The ANSI character set (value zero) is the default Windows character set. The OEM character set (value 255) is machine-specific. The number to the right of these options defines the character set. It can be any value from zero to 255, but only zero and 255 have a predefined meaning.

Font Family

These options define the family to which the font belongs. Font families define the general characteristics of the font as follows:

Family Name	Description
Roman	Proportionally-spaced fonts with serifs (Times Roman, Century Schoolbook, Bodoni, etc.)

Modern	Fixed-pitch fonts (Pica, Elite, Courier, etc.)
Swiss	Proportionally-spaced fonts without serifs (Helvetica, Univers, Swiss, etc.)
Decorative	Novelty fonts
Script	Cursive or script fonts (Old English, etc.)
Dontcare	Custom font

Italic	This option defines an italic font.
Underline	This option defines an underlined font.
Strikeout	This option defines a font whose characters have been struck out.

To make a change to this information, follow these steps:

1. Choose Header from the Font menu. The Font Editor displays the Header dialog box, which contains the font options.
2. Select the character string, number, or option you want to change.
3. For character strings and numbers, use the BACKSPACE key to delete the existing characters, then use the keyboard to type a new string or number.
4. Choose the Ok button or press the ENTER key.

8.23 Saving a Font File

You can save the changes you have made to a font by using the Save As command in the File menu. Follow these steps:

1. Choose Save As from the File menu. The Font Editor displays a dialog box that contains a text box for entering a filename.

2. Enter the name of the file in which you want save the font. This can be a new file or an existing file. Use one of the following methods:

   - Enter the font filename by typing it in the text box of the dialog box. To do this, select the box, then use the keyboard to type the name. If the box already contains a name, you can delete it by using the BACKSPACE key. Choose the Ok button or press the ENTER key.
   - If the filename you want is shown in the text box, choose the Ok button or press the ENTER key.

Once a font file is saved with your changes, you can continue editing the same font or load another font and edit it.

Note

Use the *.fnt* filename extension for all font filenames.

8.24 Editing Tips

When you are creating a new font, the closer the font you start working with is to the font you want to create, the better the results will be. For example, if you want the ANSI character set, make sure you start with an ANSI font.

You cannot change a variable-pitch font to fixed-pitch, so if you want to create a fixed-pitch font, make sure you start with a fixed-pitch font.

If you want to make several different sizes in the same kind of font, create the smallest font first. You get much better results making a small font larger than making a large font smaller.

While you are editing a font, the Font Editor keeps a copy of the font in memory. No changes to the font file are made until you save the font using the Save or Save As command.

Chapter 9
Dialog Editor

9.1 Introduction

The Microsoft Windows Dialog Editor lets you design dialog boxes on the display screen and save a definition of the box in a resource file. The definition of the dialog box can be added to other resource definitions in your application's resource script file.

When you create a dialog box, you create the box outline, put controls and text for the controls in it, and define the way the user will access the controls.

This chapter describes how to use the Dialog Editor to create and modify dialog boxes for your applications. It explains how to do the following:

- Start the editor and create the outline of a dialog box
- Add dialog controls, which the user will use to interact with the program
- Set control styles and memory-manager flags
- Define how a user can access the controls
- Edit an existing dialog box
- Create and modify the include file

Note

The Dialog Editor must be used with a mouse or similar pointing device; some keystrokes can be used, however, and these will be noted.

The Windows Dialog Editor creates dialog boxes. It does not edit other components of a resource file, such as strings, icons, and so forth. Before you use the Dialog Editor, it is a good idea to create a *.rc* (resource script) file defining those components and to use the Windows resource compiler **rc** to create a *.res* file (binary version of the resource file). You then can use the Dialog Editor to edit the *.res* file, adding dialog boxes. When you are finished creating dialog boxes, use the **#include** directive in the *.rc* file to name the include file for the dialog boxes. Use the **rcinclude** keyword to name the *.dlg* files.

For more information about the resource script file, see Chapter 3, "Resource Compiler: Rc." For more information about the *.dlg* and *.res* files, see Section 9.10.

9.2 Starting the Editor

To start the Dialog Editor, open the MS-DOS Executive window and double-click the filename *dialog.exe*. Windows loads the Dialog Editor and displays the window shown in Figure 9.1:

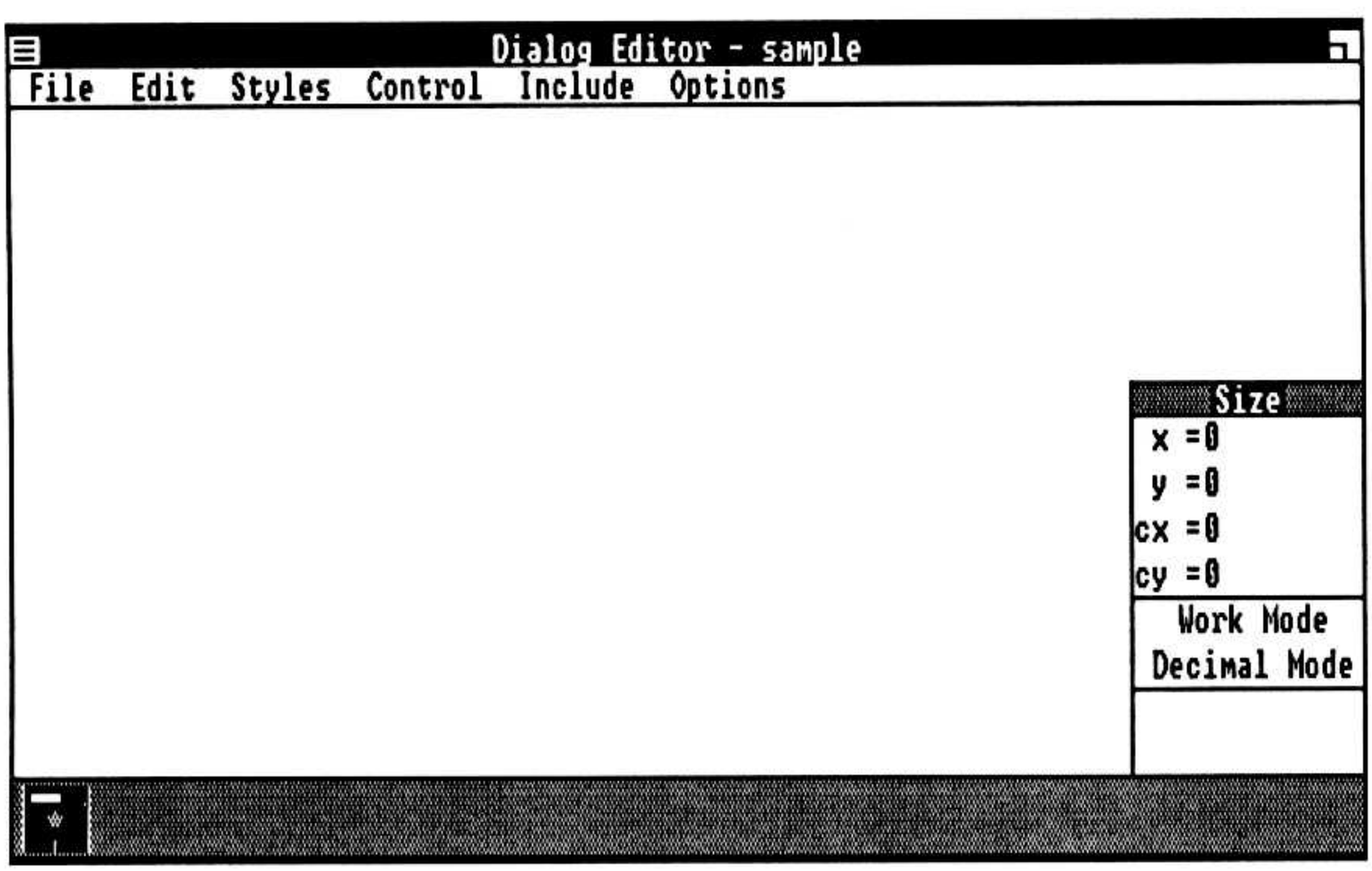

Figure 9.1 Dialog Editor Window

The Dialog Editor's menu bar contains the following menus:

Menu	Contents
File	Commands that create, open, and save the files containing dialog boxes. There is also a command that allows you to view existing dialog boxes.
Edit	Commands that allow you to perform common editing functions such as cutting and pasting. There are also commands for creating a new dialog box, renaming an existing one, and defining the units by which the mouse moves.
Styles	Commands that allow you to control the styles, text, and ID values for the dialog-box controls. There is also a command for defining memory management.
Control	Commands that let you define the type of controls to be placed in the dialog box.
Include	Commands that you use to create, modify, or view an include file.
Options	Commands that allow you to define the order in which controls are accessed, and a command that allows you to test your dialog box.

9.3 Using the Size Window

When you start the Dialog Editor, you will notice a small window labeled "Size" in the lower-right corner of the screen. The Size window stays on your screen as you edit a dialog box and supplies you with information about the dialog box and the controls in it. When you make a change to the dialog box or controls, the change is reflected in the Size window. If necessary, the Size window can be moved out of the way of a dialog box you are working on. Figure 9.2 illustrates the Size window.

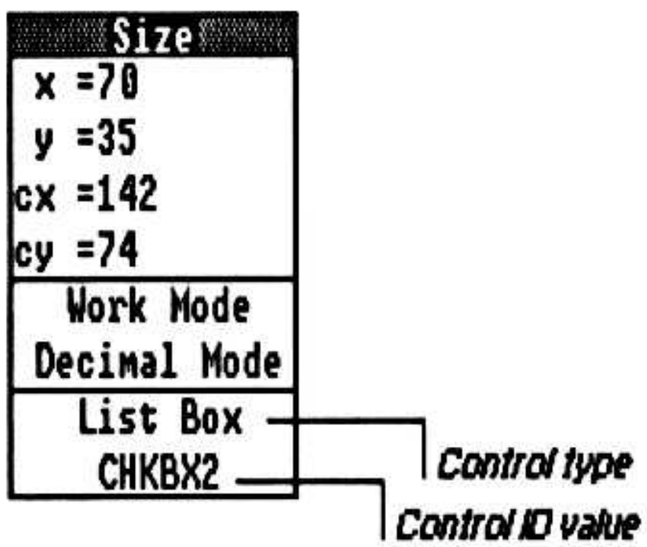

Figure 9.2 Size Window

The Size window displays the information shown in the following list. All size/position values are in dialog units. (A dialog unit is a horizontal or vertical distance. One horizontal dialog unit is equal to 1/4 the width of a character in the system font. One vertical dialog unit is equal to 1/8 the height of a character in the system font.)

Field	Description
x	Displays the position on the *x*-axis (vertical position) of the upper-left corner of the dialog box or control you have selected.
y	Displays the position on the *y*-axis (horizontal position) of the upper-left corner of the dialog box or control you have selected.
cx	Displays the height of the dialog box or control you have selected.
cy	Displays the width of the dialog box or control you have selected.
Work/Test Mode	Indicates whether the Dialog Editor is in Work Mode, in which you can edit the dialog box, or Test Mode, in which you can try out the controls in the dialog box.
Decimal/Hex Mode	Indicates whether the ID values for the controls are shown in decimal or hexadecimal numbers.
Control Type	Shows the type of control you have selected (for example, Radio Button or Check Box). If the dialog box was selected, this part of the Size window will read "Dialog."

Control ID Value	Shows the ID value of the control you have selected. If the dialog box was selected, no ID value is shown.

9.4 Creating a Dialog Box

The first step in creating a dialog box is to create and size the outline of the box.

9.4.1 Clearing the Display

You should always clear the Dialog Editor's display before starting a new dialog box. To clear the display, choose the New command from the File menu.

The Dialog Editor clears the display, removing any existing dialog box. If you have previously made changes to the box, you will see a dialog box asking whether you want to save the changes. The New command opens a new file called *sample*, which initially is empty.

9.4.2 Drawing the Border

To create the border for a dialog box, use the Edit menu. Follow these steps:

1. Choose the New Dialog command. You will be asked to enter a name for the new dialog box.
2. Type a name for the box.
3. Choose the Ok button or press the ENTER key. This puts an empty dialog box on your screen.

9.4.3 Expanding/Shrinking a Dialog Box

To increase or decrease the size of the dialog box, use one of the eight "handles" (small, filled rectangles) on the boundaries, as shown in Figure 9.3.

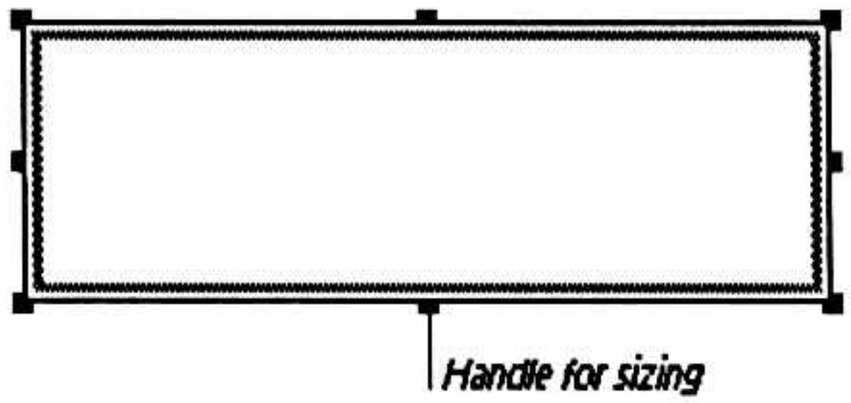

Figure 9.3 Outline of a Dialog Box

To change the size of a dialog box, follow these steps:

1. Select the dialog box.
2. Move the mouse cursor to a handle on the side you want to move. The cursor will change to a small box (similar to the handle.)
3. With the mouse button, drag the border in the desired direction. When you release the mouse button, the dialog box will retain its new border. You can size the box in vertical and horizontal directions simultaneously by using a corner handle.

9.5 Adding and Deleting Controls

Controls in a dialog box allow the user to interact with the application. Once you have created the border for the dialog box, you can enter any number of the following controls:

Control	Action
Check Box	Creates a check box, a small square with a label to its right. Check boxes typically are used in groups to give the user a choice of selections, any number of which can be turned on or off at a given moment.
Radio Button	Creates a radio button, a small circle with a label to its right. Radio buttons typically are used in groups to give the user a choice of selections, only one of which can be selected at a time.

Push Button	Creates a push button, a small, rounded rectangle that contains a label. Push buttons are used to let the user choose an immediate action, such as canceling the dialog box.
Group Box	Creates a simple rectangle that has a label on its upper edge. Group boxes are used to enclose a collection or group of other controls, such as a group of radio buttons.
Horizontal Scroll Bar	Creates a horizontal scroll bar. Scroll bars let the user scroll data and usually are associated with another control or window that contains text or graphics.
Vertical Scroll Bar	Creates a vertical scroll bar.
List Box	Creates a simple rectangle that has a vertical scroll bar on its right edge. List boxes are used to display a list of strings, such as file or directory names.
Edit	Creates an edit control, a rectangle in which the user can enter and edit text. Edit controls are used both to display numbers and text and to let the user type in numbers and text.
Text	Creates a static text control. Static text controls are used as labels for other controls, such as edit controls.
Frame	Creates a rectangle that you can use to frame a control or group of controls.
Rectangle	Creates a filled rectangle.
Icon	Creates a rectangular space in which you can place an icon. (Do not size the icon space; icons automatically size themselves.)

9.5.1 Adding Controls

To add controls to a dialog box, use the Control menu. Follow these steps:

1. Select the control you want. The mouse cursor changes to a plus sign (+).

2. Position the cursor where you want to place the control.
3. Press the mouse button. The control appears in the dialog box. If it has text associated with it, the word "text" is included. To add text, see Section 9.5.2.

9.5.2 Adding Text to Controls

To add or change the text in a control, follow these steps:

1. Select the control.
2. Choose Class Styles from the Styles menu. You will see a dialog box showing style information about the control.
3. In the Window Text section, type the text you want to appear in the control. If you want to edit what you have typed, use the direction arrows to scroll the text. Text can be inserted at the cursor. To delete characters, use the BACKSPACE key.
4. Choose the Ok button or press the ENTER key to confirm your entry. (The Class Styles dialog box has other uses, described in Section 9.6.1.)

Note

When you add an Icon control to a dialog box, the text should be the name that was defined for the icon in the *.rc* file. For example, assume the *.rc* file contains the following entry:

```
myicon icon my.ico
```

To use the icon in a dialog box, you create an Icon control and type the name "myicon" in the Window Text section.

9.5.3 Moving a Control

You can reposition a control in a dialog box either by using the mouse to drag it to a new location or by using the DIRECTION keys for fine adjustments. To move a control, follow these steps:

1. Select the control. The mouse cursor changes to a plus sign (+).
2. Drag the control to its new location.

To move a control one dialog unit at a time, use the DIRECTION keys. In this way, you can move a control a few positions over (or up or down) without affecting its position on the other axis. This is helpful when you want to line up the controls.

9.5.4 Moving a Group of Controls

You can move a group of controls from one location in a dialog box to another. This can be useful if you decide to rearrange the layout of controls in the box and you have two or more controls that you want to keep together. To move a group of controls, follow these steps:

1. Press and hold the CONTROL key.
2. While pressing the CONTROL key, select each control you want to keep in the group by using the mouse button. Each control will be outlined with a gray line; the group of controls will also have a gray border around it. (If you change your mind, you can reverse a selection by clicking it with the mouse button. You must still be pressing the CONTROL key.)
3. While still pressing the CONTROL key, point the mouse cursor at a location inside the group border, but not inside any of the controls' borders, as shown in Figure 9.4:

Figure 9.4 Cursor Position for Moving a Group of Controls

4. Press the mouse button, then release the CONTROL key.
5. Drag the group of controls to the desired location and release the mouse button. The group of controls is placed in the new location.

9.5.5 Changing a Control's Size

To increase or decrease the size of a control, use one of the eight handles (small rectangles) on the boundaries. Follow these steps:

1. Select the control.
2. Move the mouse cursor to a handle. The cursor will change to a small box, similar to the handle.
3. Drag the border in the desired direction. When you release the mouse button, the control boundary will retain the new size.

9.5.6 Deleting Controls

To delete a control from a dialog box, follow these steps:

1. Select the control.
2. Choose Clear Control from the Edit menu. The control will be deleted.

9.6 Changing Control Styles and Memory-Manager Flags

The Styles menu allows you to set control styles and attributes for memory-manager flags.

Control styles dictate such things as whether a control can be grayed, or whether a button is a default push button. Control styles available to a given class of controls are set by using the Class Styles command. Control styles available to all controls and to the dialog box itself are set by using the Standard Styles command.

Memory-manager flags determine whether a code segment is moveable or discardable and if it will be preloaded.

9.6.1 Changing Class Styles

The Class Styles command in the Styles menu allows you to change the control styles that govern a control. You can also use this command to enter or change text in a control and to change the control's ID value. (If an include file was loaded, you may symbolically refer to the control's ID value. For more information on include files, see Section 9.10.1.)

Figure 9.5 shows the Class Styles dialog box for a button control.

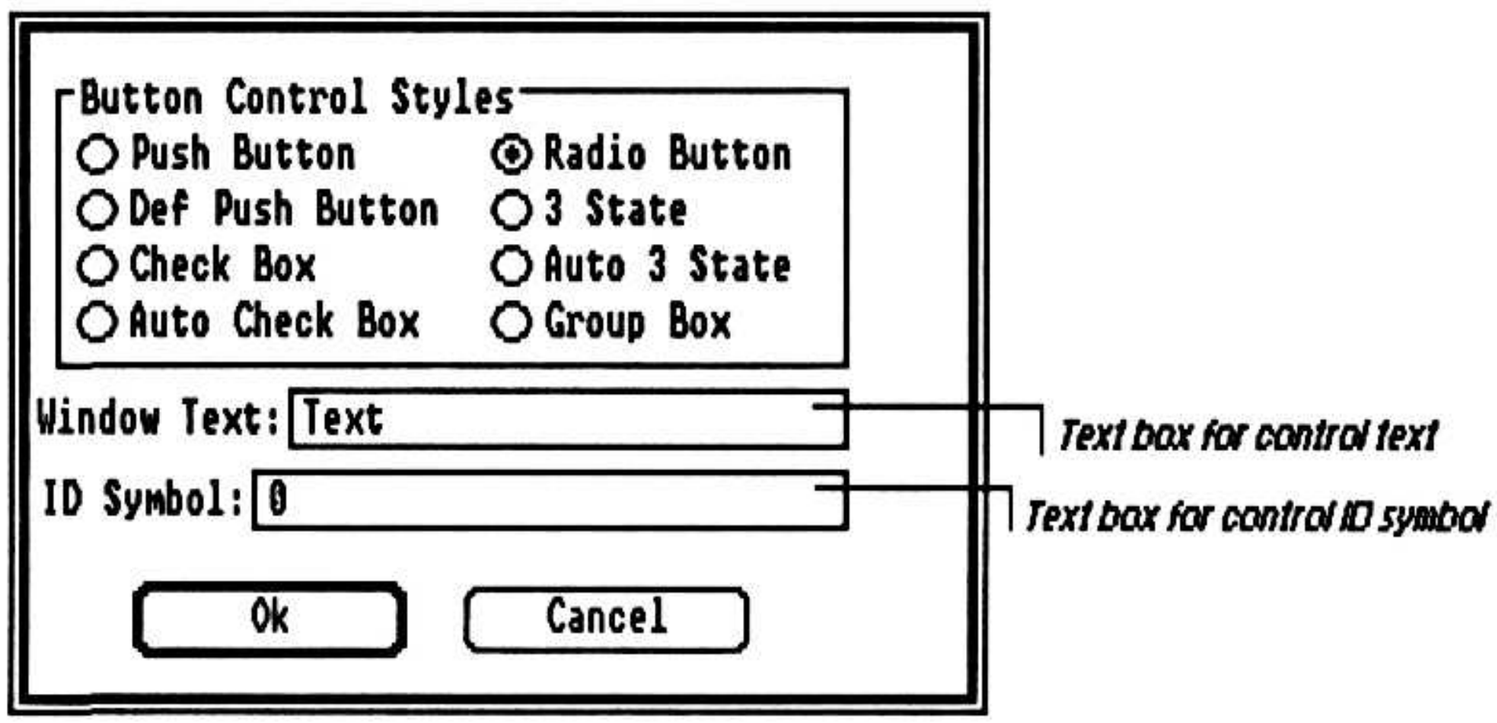

Figure 9.5 Button Control Styles Dialog Box

To change a control style for a specific control, follow these steps:

1. Select the control.
2. Choose Class Styles from the Styles menu. You will see a dialog box that relates to the control you selected.
3. Select the desired options. Control-style options are described in the *Microsoft Windows Programmer's Reference.*
4. Choose the Ok button or press the ENTER key.

9.6.2 Changing Standard Styles

The Standard Styles command in the Styles menu generates a dialog box that lists control styles available to all controls and to the dialog box itself. These are known as global window styles. The Standard Styles dialog box is shown in Figure 9.6. You can use the Standard Styles command to do such things as add a group marker or tab stop for a control.

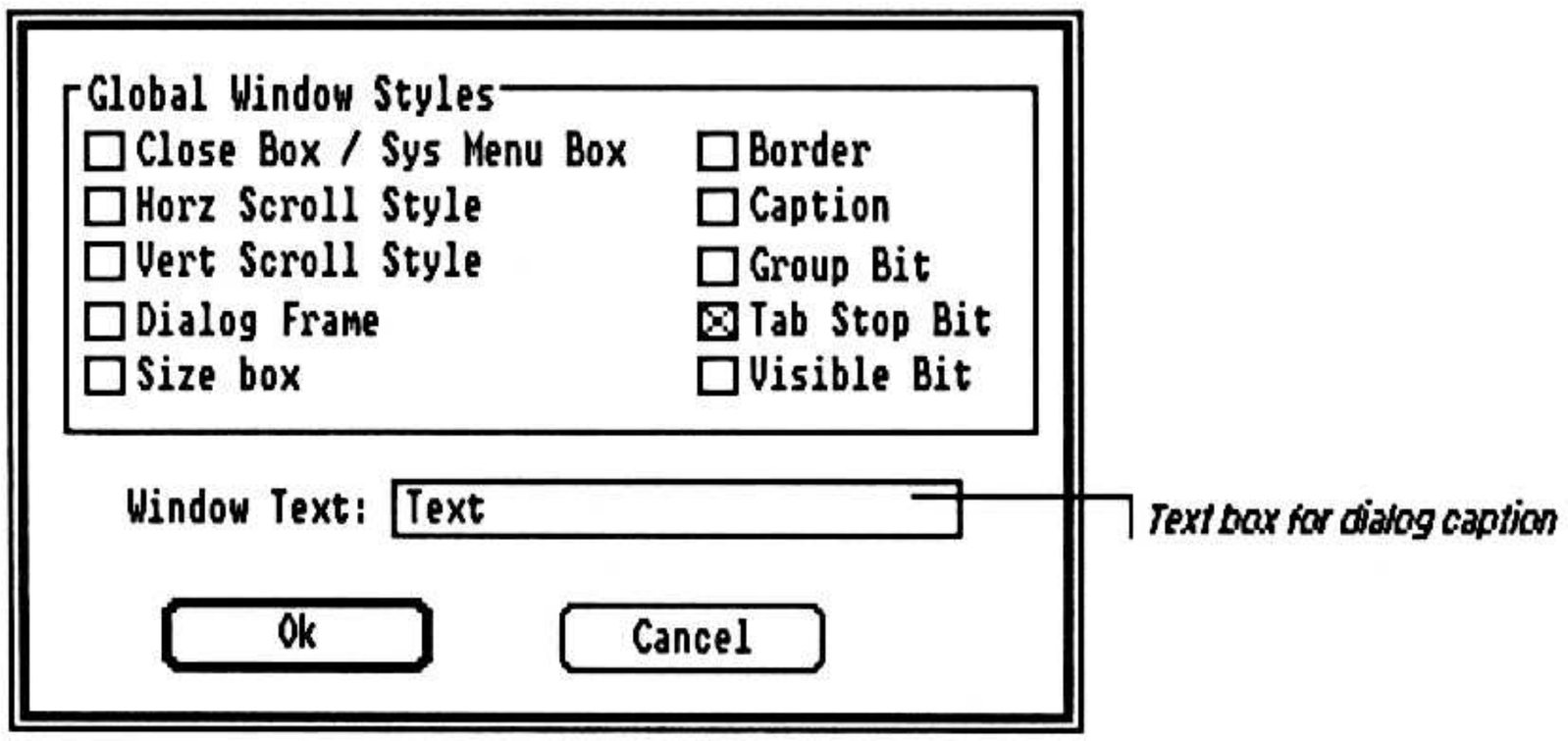

Figure 9.6 Standard Styles Dialog Box

To set global window styles, follow these steps:

1. Select the appropriate control or dialog box. A dialog box will appear.
2. Turn on the style check boxes you want to add (or turn off the ones you want to delete).
3. Add or delete text in the Window Text section if desired.
4. Choose the Ok button or press the ENTER key.

Note

If the Visible Bit check box is turned on, the dialog box will be displayed whenever it is called by the program. This may not be desirable in some cases. For example, if you have a command with an accelerator, you may not want to display a dialog box when the accelerator is used. You can prevent the display of a dialog box by leaving the Visible Bit check box turned off. Generally, it is best to leave the Visible Bit check box turned off unless you want the dialog box to be seen in all cases.

9.6.3 Including a System Menu Box, Size Box, or Scroll Bars

You can include a System-menu box, size box, or vertical or horizontal scroll bars as part of a dialog box. (These scroll-bar options are part of the dialog box; they are not separate controls. For more information on scroll-bar controls, scrolling functions, or window messages resulting from scrolling, see the *Microsoft Windows Programmer's Reference.*) For example, a modeless dialog box must have a System menu so the user can close the box.

To include a System-menu box, size box, or scroll bars in a dialog box, follow these steps:

1. Select the dialog box.
2. Choose Standard Styles from the Styles menu.
3. Turn on the appropriate check box(es).
4. Turn on the Border check box. (Each of these options requires that you turn on the Border check box. When you do so, the Caption check box is automatically turned on.)
5. Choose the Ok button or press the ENTER key to confirm your selections.

9.6.4 Setting Memory-Manager Flags

Memory-manager flags determine how the code for a dialog box is treated by the application and by Windows with regard to memory. You can set options to specify when a resource is to be loaded into memory, as well as whether the resource is fixed, moveable and/or discardable. Memory-manager flags are set in the dialog box shown in Figure 9.7.

Figure 9.7 Resource Properties Dialog Box

To set these flags, follow these steps:

1. Select the dialog box.
2. Choose the Resource Properties command from the Styles menu. You will see the dialog box shown in Figure 9.7.
3. Turn on (or off) the check boxes corresponding to the memory-manager flags you want. (These options are described in the *Microsoft Windows Programmer's Reference.*)
4. Choose the Ok button or press the ENTER key.

9.7 Defining User Access to Controls

The way a dialog box reacts to the keyboard or mouse interface is based in part on the sequential order of the controls and the location of tab stops. These options are set with the Order Groups command from the Options menu. Using this command, you can define the following:

- The sequential order of the controls.
- Which *groups* the controls are in and the sequential order of the groups. (A group is a collection of controls. Within a group of controls, the user makes selections using the DIRECTION keys.)
- The location of tab stops (the place where the cursor moves when the user presses the TAB key).

When you choose the Order Groups command, you will see the dialog box shown in Figure 9.8.

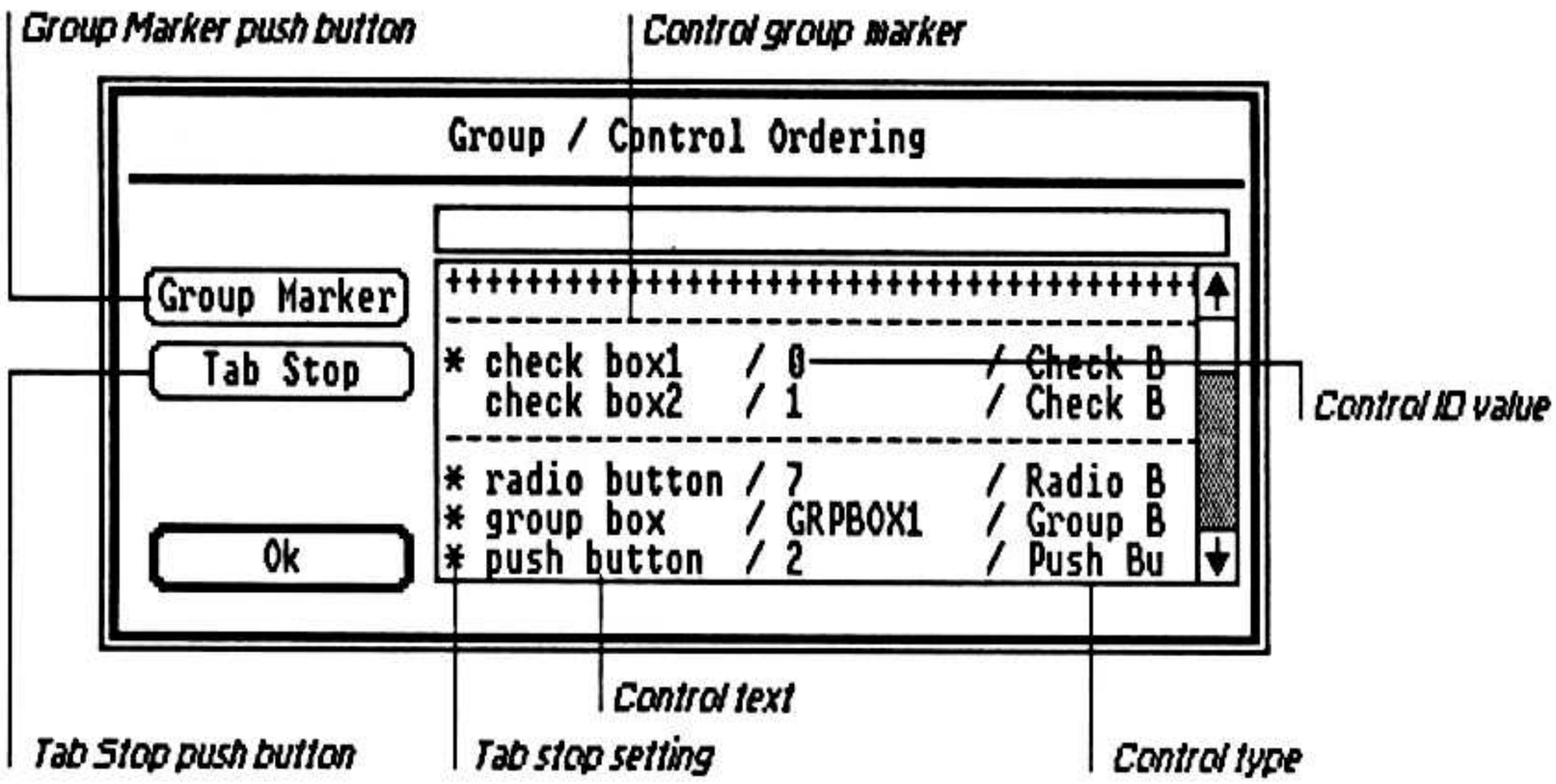

Figure 9.8 Group/Control Ordering Dialog Box

9.7.1 Changing the Order of Controls

By default, the controls you place in a dialog box receive the input focus (and thus are accessed by the user) in the order in which they were placed in the box. For example, the first control you put in the box will receive the focus first, no matter where you subsequently move it in the dialog box. To change the sequential order, you must use the Order Groups command and rearrange the controls in the list it displays.

When you rearrange the order of the controls in the Group/Control Ordering dialog box, the control statements in the *.dlg* file are rearranged correspondingly. Thus, the first control listed in the *.dlg* file is the first to receive the input focus, the second listed is the second to receive the focus, and so on.

To change the sequence of a control in a dialog box, follow these steps:

1. Choose Order Groups from the Options menu. You will see the dialog box shown in the preceding Figure 9.8.
2. From the list in the dialog box, select the control you want to move.
3. Place the mouse cursor where you want the control to appear. Notice that as you move it, the cursor changes from an arrow to a short, horizontal bar. The bar appears only in places where you are allowed to insert the control.
4. To insert the control, press the mouse button.

Note

If you decide to choose a different control to move, you can do so whenever the pointer appears as an arrow. Just point to the desired control and press the mouse button. The new selection will replace the old.

9.7.2 Setting a Tab Stop

Tab stops determine where the cursor will move when the user presses the TAB key. Normally, tab stops are set for individual controls or, in the case of a group, for the first control in the group. To set a tab stop, follow these steps:

1. Choose Order Groups from the Options menu.
2. Select the control at which you want to place the tab stop.
3. Select the Tab Stop button.
4. Choose the Ok button. An asterisk appears next to the control, which indicates a tab stop has been placed.

Note

You can also set a tab stop by selecting the control, choosing Standard Styles from the Styles menu, and then turning on the Tab Stop Bit check box.

9.7.3 Deleting a Tab Stop

To delete a tab stop, follow these steps:

1. Choose Order Groups from the Options menu.
2. Select the control that has the tab stop. The Tab Stop button will change to read "Delete Tab."
3. Select the Delete Tab button. The asterisk disappears.
4. Choose the Ok button or press the ENTER key.

Note

You can also delete a tab stop by selecting the control, choosing Standard Styles from the Styles menu, and turning off the Tab Stop Bit check box.

9.7.4 Adding a Group Marker

To designate the beginning and end of a group, you add a *group marker* to the list of controls in the group. (The group marker appears in the Group Order dialog box as a horizontal dashed line, as shown in the preceding Figure 9.8). You need to place a group marker both before the first control and *after* the last control in a group. To add a group marker, follow these steps:

1. Choose Order Groups from the Options menu.
2. Select the control that appears just below where you want to place the group marker.
3. Select the Group Marker button. The horizontal line indicates that the group marker has been inserted.
4. Repeat steps 2 and 3 until all markers have been placed.
5. Choose the Ok button.

Note

You can also add a group marker using the Standard Styles command from the Styles menu. Select the control, then turn on the Group Bit check box. A group marker will be placed just above the control you selected.

9.7.5 Deleting a Group Marker

To delete a group marker, follow these steps:

1. Choose Order Groups from the Options menu.
2. Select the group marker line. The Group Marker button will change to read "Delete Marker."
3. Select the Delete Marker button.
4. Choose the Ok button.

Note

You can also delete a group marker by selecting the control, choosing Standard Styles from the Styles menu, and turning off the Group Bit check box.

9.8 Modifying a Dialog Box

To modify an existing dialog box, you open the *.res* file containing the dialog box, then use the procedures listed in this chapter to make the changes. To open the *.res* file and the dialog box for editing, follow these steps:

1. Choose Open from the File menu. You will see a dialog box listing the available *.res* files in the current directory. The *.res* files contain the dialog boxes. If the *.res* file you want is in another directory, type the pathname of the directory (or drive) in the text box.
2. Open the appropriate *.res* file. You will see a dialog box listing the available include files (*.h* extension).
3. If you want to open an include file, do so. If not, choose the Cancel button. You will see a dialog box listing the dialog boxes in the file.
4. Open the desired dialog box by double-clicking the name or by selecting the name and choosing the Ok button. The dialog box you choose will appear on the screen and you can begin editing it.

If you want to work on another dialog box in the same file, choose View Dialog from the File menu. You again will see the list of dialog names you can choose from.

9.9 Using the Edit Menu

The commands in the Edit menu will help you create or modify dialog boxes. The following list shows the command names and the result of each command:

Command	Result
Restore Dialog	Allows you to restore the dialog box to its previous saved state.
Cut Dialog	Deletes the currently displayed dialog box and puts it in the Clipboard. (It cuts both the dialog format and the bitmap format, both of which can be edited with Paint.)
Copy Dialog	Puts a copy of the currently displayed dialog box (both the dialog format and the bitmap format) in the Clipboard.
Paste Dialog	Puts the contents of the Clipboard on the screen if the contents are in dialog format.
Clear Dialog/Control	Deletes the currently selected dialog box or control. If it is a dialog box, a confirmation message will be displayed.
New Dialog	Puts the currently displayed dialog box back into the *.res* file and places a new, empty dialog box on the screen. Requests the name of the new dialog box.
Rename Dialog	Requests a new name for the dialog box currently displayed.
Grid	Determines the location of the upper-left corner of a control. The grid is defined in multiples of dialog units. For example, when the grid is set at 20 horizontal (dx) units and 20 vertical (dy) units, the numbers defining the position of the control's upper-left corner will be in multiples of 20. Default settings are one dialog unit each in both horizontal and vertical directions.

9.10 Using Files with the Dialog Editor

The menus and strings that make up the user interface for a Windows application generally are defined in the resource script file, a text file that has the *.rc* extension. The application's dialog boxes are defined in a text file that has the *.dlg* extension. These files are processed by **rc**, the Windows resource compiler, producing a binary resource file that has the *.res* extension. Ultimately, this *.res* file is linked to the application's executable *.exe* file.

When the Dialog Editor is used, the *.res* file containing the binary form of definitions for the dialog boxes is modified. A typical use of the Dialog Editor would be to open a *.res* file, create or modify dialog boxes, then save the results. Saving your results overwrites both the original *.res* file and any existing *.dlg* files. The new *.res* file contains the new dialog definitions, plus everything else that was included in the original *.res* file. This new *.res* file can be linked immediately to the application source file. The new *.dlg* file is created as a backup in case you ever need to re-create the *.res* file using **rc**. (For example, if you wanted to change or add to the menus and strings in the *.rc* file, you would have to do this manually, so the resource file would need to be recompiled.) The *.dlg* file must be put into the *.rc* file, either manually or by using the **rcinclude** keyword, before recompiling.

One additional file, the include file, is associated with the dialog boxes created using the Dialog Editor. The include file is described in Section 9.10.1.

9.10.1 Include File

The include file contains control ID definitions (#**define** directives). These are used to define internal control numbers as symbolic constants. The symbolic constants then can be used in the application source code. For example, if a dialog box contains an Ok push button, you can define the button ID as a constant, giving it a symbolic name such as CTLOK. Then you can use the symbolic name, CTLOK, in your application source code.

When assigning ID values to controls, you can assign any numbers you want; however, there are some guidelines you should keep in mind. To use the ENTER and ESCAPE keys in standard ways, you need to create an include file and assign meaningful values to the corresponding controls. ID numbers 1 and 2 have special meanings. You should use these numbers as described in the following paragraphs so as not to confuse the user with

controls that do not respond as expected. If you decide not to follow these guidelines, you should not use ID values of 1 and 2.

ID Value of 1

When the ENTER, CANCEL, or ESCAPE key is pressed, Windows automatically sends a response message to the dialog input function. If the dialog box has a default button (for example, the Ok button), pressing the ENTER key sends a WM_COMMAND message, along with the ID value of 1. Thus, if you have a default Ok button, you should assign it an ID value of 1 so it will be activated when the user presses the ENTER key. This is consistent with the recommended guidelines for creating a Windows application. (For information on application guidelines, see the *Microsoft Windows Application Style Guide.*)

ID Value of 2

When the CANCEL or ESCAPE key is pressed, Windows automatically sends a WM_COMMAND message, along with the ID value of 2. Thus, if you have a Cancel button in a dialog box, it should have an ID value of 2.

9.10.2 Creating an Include File

To create an include file, follow these steps:

1. Select the control you want to define in the file.
2. Choose View Include from the Include menu. You will see a dialog box similar to the one shown in Figure 9.9.

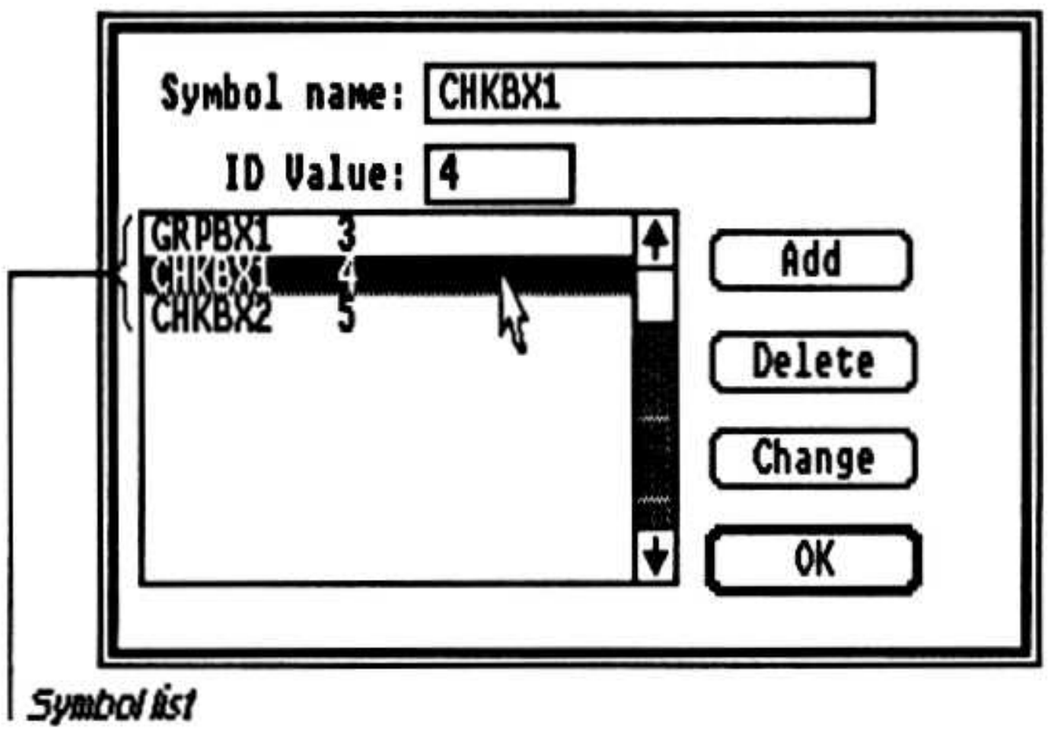

Figure 9.9 View Include Dialog Box

3. In the Symbol name text box, type the symbolic name you are giving to the control ID.
4. In the ID Value text box, type the number you are assigning as the ID value.
5. Select the Add button.
6. Choose the Ok button or press the ENTER key.
7. Choose Save from the Include menu.

9.10.3 Editing an Include File

You can make changes to the symbols listed in an include file by using the View Include command from the Include menu. For example, you may want to change the name of a symbol or delete one of the symbols. To edit the include file, follow these steps:

1. Choose View Include from the Include menu. You will see a dialog box that lists the symbols in the file.
2. Select the symbol you want to change or delete.
3. To change a symbol's name or ID value, make the change in the appropriate list box, then select the Change button. To delete the symbol, select the Delete button.
4. Choose the Ok button or press the ENTER key.

9.11 Saving a Dialog Box

Once you have created or made changes to a dialog box, save the changes by choosing Save from the File menu. By default, the file will have the name *sample.res*. If you want to give it a different name, choose Save As and enter the new name in the resulting dialog box.

Chapter 10
Shaker and Heapwalker

10.1 Introduction

The Microsoft Windows Shaker and Heapwalker applications let you examine different aspects of system memory and see the effect your application has on them. This chapter explains how to use the Shaker and Heapwalker applications.

10.2 Testing Moveable Memory: Shaker

The Shaker application lets you see the effect of memory movement on your application. The Shaker randomly allocates and frees chunks of global memory with the intention of forcing the system to move moveable data or code segments in your application. The Shaker is useful for making sure your application properly locks code and data segments when attempting to access them.

To start the Shaker, open the MS-DOS Executive window and double-click the filename *shaker.exe*. Windows loads the application and displays the Shaker window. Figure 10.1 shows the screen that is displayed when the On command is chosen from the Show State window:

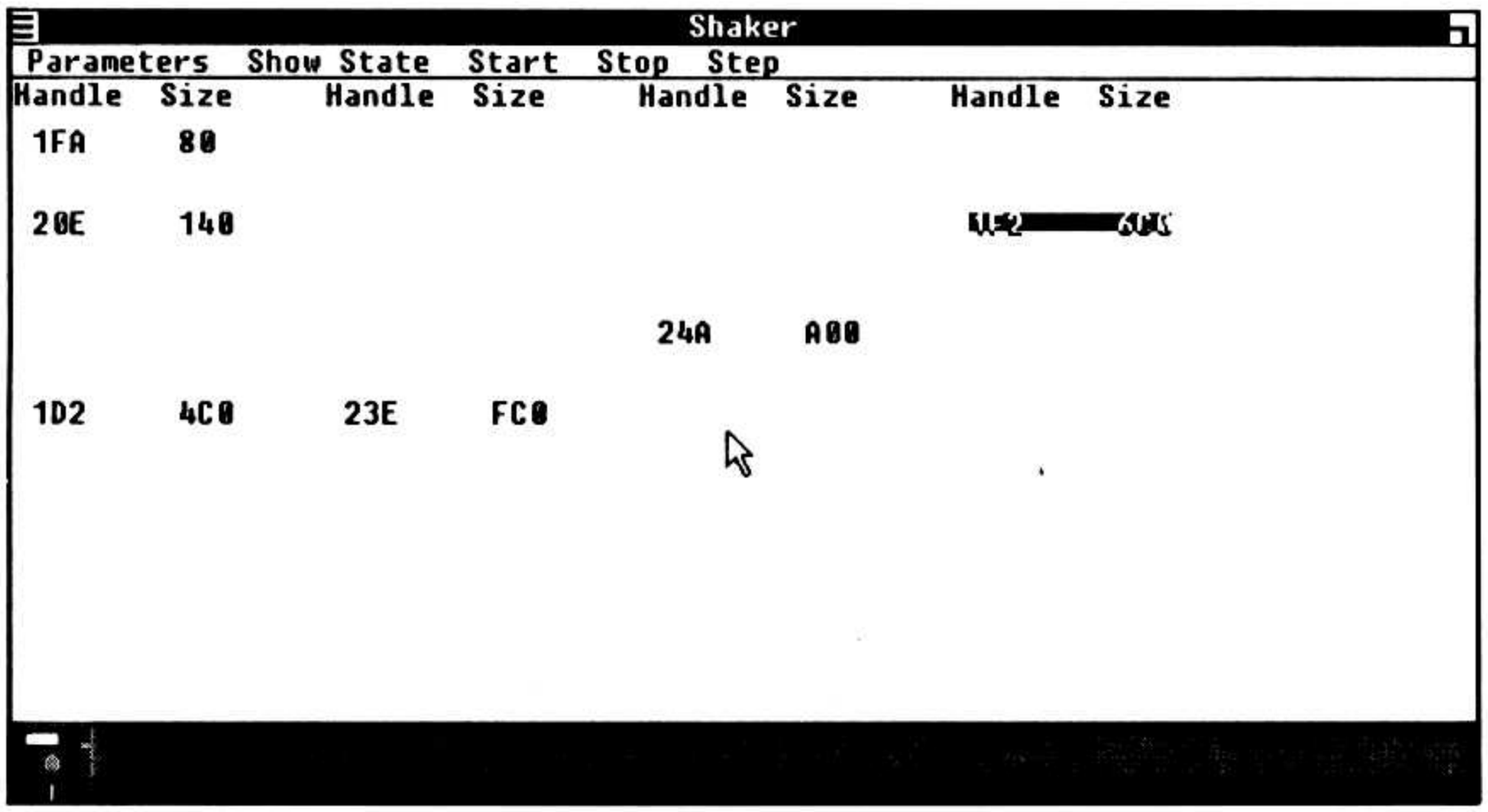

Figure 10.1 Shaker Window with Show State On

The Shaker has the following commands:

Command	Action
Allocation Granularity (in Parameters menu)	Sets the minimum size of the objects to be allocated. Each object is some multiple of this size; for example, if the granularity is 128, the Shaker allocates objects that have byte sizes of 128, 256, 384, and so on. The smaller the granularity, the more likely it is that the allocated objects will fit in the spaces between global objects.
Time Interval (in Parameters menu)	Sets the time interval, in system-timer ticks, between allocations. The Shaker allocates a new object after every interval elaspes. If the maximum number of objects has been allocated, it reallocates one it has already allocated.
Max Objects (in Parameters menu)	Sets the maximum number of objects to be allocated.
On (in Show State menu)	Displays the object handles and the allocation sizes.
Off (in Show State menu)	Removes display of object handles and allocation sizes.
Start	Starts the allocation.
Stop	Stops the allocation.
Step	Allocates one object and stops. This command can be used when the Shaker otherwise is stopped.

10.3 Viewing the Global Heap: Heapwalker

The Heapwalker application lets you examine the global heap. It displays information about all objects in system memory and is useful for seeing what effect your application has when allocating memory for its own use.

To start the Heapwalker, open the MS-DOS Executive window and double-click the filename *heapwalk.exe*. Windows loads the application and displays the Heapwalker window. Figure 10.2 shows the screen that is displayed when the Walk command is chosen from the Walk menu.

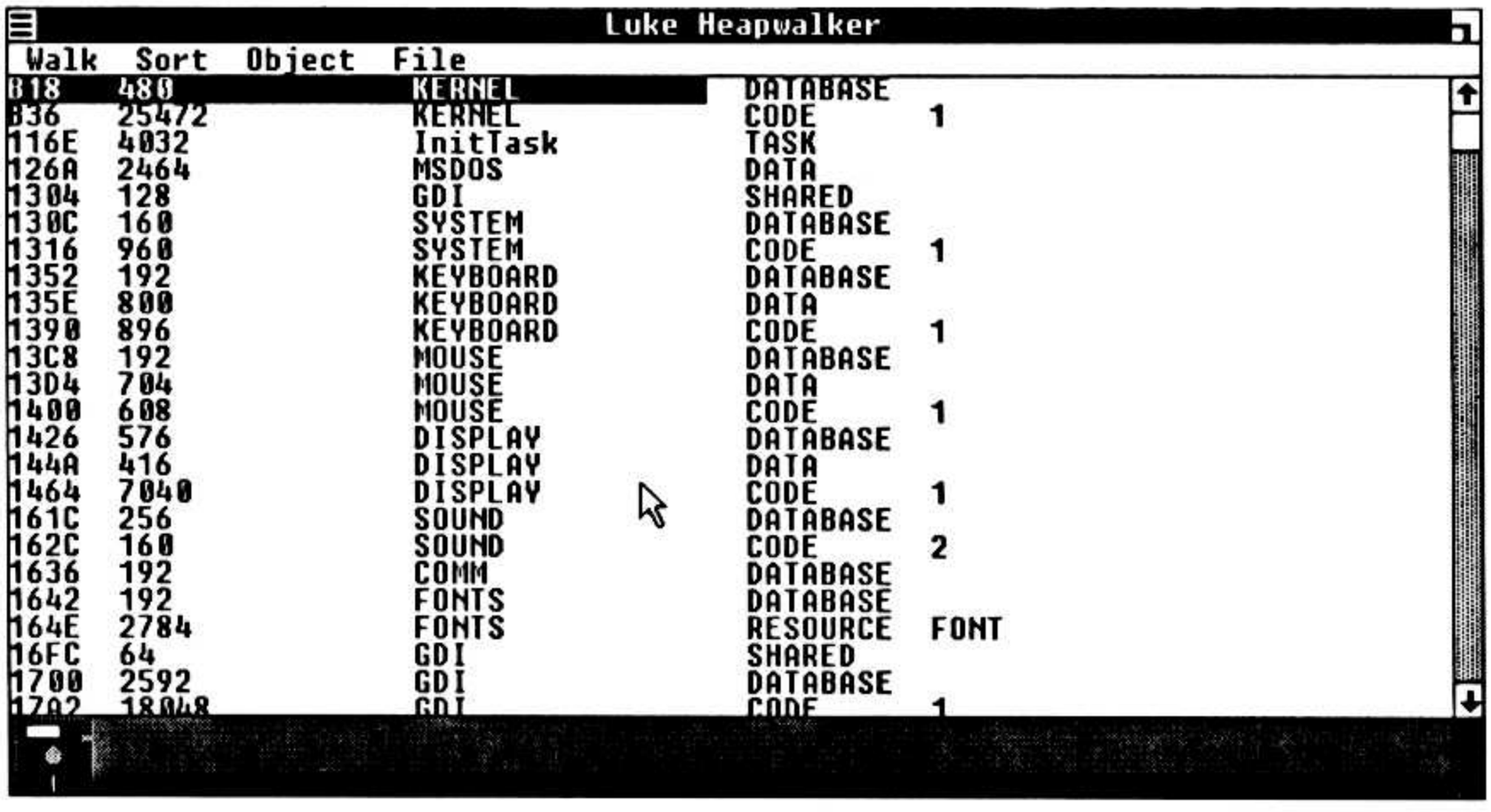

Figure 10.2 Heapwalker Window after Walk Command

The global heap consists of all of available system memory. The heap contains global objects, areas of memory that have been allocated for some specific use. Some of these objects are free and can be allocated the next time an application calls the **GlobalAlloc** or **GlobalRealloc** function. Some of the objects have been allocated and contain data segments, code segments, resources, etc.

The Heapwalker application has the following commands:

Command	Action
Walk (in Walk menu)	Displays the current state of memory and identifies each object. Each display line identifies one global object. The display shows the following: 1. The segment address of the object (actually the segment of the arena header; the object starts one paragraph later) 2. The size of the object in bytes 3. The lock count (for example, L2) 4. Discardable flag, D 5. The object's owner 6. The object type (code, data, resource, shared) 7. Additional information for that object (segment number for code, type of resource)
GC(0) and Walk (in Walk menu)	Performs a global compact, asking for zero bytes, then displays the heap.
GC(–1) and Walk (in Walk menu)	Allocates all of memory (causing all discardable objects to be thrown out), then displays the heap.
GC(–1) and Hit A: (in Walk menu)	Used for internal testing.
Allocate all of memory (in Walk menu)	Allocates all of free memory, which is useful for testing out-of-memory conditions in applications.
Free allocated memory (in Walk menu)	Frees the memory allocated by the Allocate all of memory command.
Free 1K of allocated memory (in Walk menu)	Frees 1 kilobyte of memory allocated by the Allocate all of memory command. It is used to approach out-of-memory conditions.

Address (in Sort menu)	Sorts the display by address.
Module (in Sort menu)	Sorts the display by module name.
Size (in Sort menu)	Sorts the display by allocation size.
Show (in Object menu)	Displays the contents of the selected object in hexadecimal and ASCII. An object can be selected by clicking the object display.
LocalWalk (in Object menu)	Displays the objects in the currently selected local heap (data object). This display shows the following: • The offset in the **ds** register of the object. • The size in bytes of the object. • Allocation status (Busy or Free). • The object type (Fixed or Moveable). The first object in the local heap is allocated by the memory manager, so there are always at least two objects in a local heap.
LC(–1) and LocalWalk (in Object menu)	Compacts the selected local heap, then displays the heap.
Write (in File menu)	Displays the global heap and writes the results to a file named *heapwalk.out*.

Appendixes

Appendix A
Disk Contents

A.1 Disk Contents

This appendix contains a list of the files provided in the Microsoft Windows Software Development Kit.

A.1.1 Windows Development Utilities

File	Purpose
exehdr.exe	Decodes and displays the file header in executable Windows applications.
implib.exe	Creates linkable library files for user-defined, dynamic Windows libraries.
lib.exe	Creates and maintains library (*.lib*) files.
link4.exe	Creates executable Windows applications.
make.exe	Performs automatic file maintenance based on a **make** file.
mapsym.exe	Creates symbol files for symbolic debugging.
rc.exe	Compiles Windows resources defined in a resource script file and adds them to Windows applications.
rcpp.exe	Preprocesses resource script files.
symdeb.exe	Symbolically debugs Windows applications.
winstub.exe	Displays a Windows-required warning message. This executable file typically is placed at the beginning of an executable Windows application to prevent the application from being executed in a non-Windows environment.

A.1.2 Windows Development Applications

File	Purpose
atrm1111.fnt	Sample font file for use with *fontedit.exe.*
dialog.exe	Creates dialog boxes for Windows applications.

fontedit.exe	Creates font files for Windows applications.
heapwalk.exe	Displays list of owners and sizes of allocated blocks in Window's global heap.
iconedit.exe	Creates icons, cursors, and bitmaps for Windows applications.
shaker.exe	Randomly allocates memory in the global heap to force movement.

A.1.3 Windows Sample Applications: C Language

File	Purpose
clock	**Make** file for *clock.exe*
clock.c	Source file for *clock.exe*
clock.def	Module definition file for *clock.exe*
clock.h	Include file for *clock.exe*
clock.ico	Icon file for *clock.exe*
clock.rc	Resource script file for *clock.exe*
clockdat.asm	Data file for *clock.exe*
comm.c	Source file for *terminal.exe*
declare.h	Include file for *sample.exe*
dlgopen.c	Source file for *sample.exe*
dlgsave.c	Source file for *sample.exe*
fonttest	**Make** file for *fonttest.exe*
fonttest.c	Source file for *fonttest.exe*
fonttest.def	Module definition file for *fonttest.exe*
fonttest.h	Include file for *fonttest.exe*
fonttest.ico	Icon file for *fonttest.exe*
fonttest.rc	Resource script file for *fonttest.exe*
gettime.asm	Assembly source file for *clock.exe*
hello	**Make** file for *hello.exe*

hello.c	Source file for *hello.exe*
hello.def	Module definition file for *hello.def*
hello.h	Include file for *hello.exe*
hello.ico	Icon file for *hello.exe*
hello.rc	Resource script file for *hello.rc*
mapmodes	**Make** file for *mapmodes.exe*
mapmodes.c	Source file for *mapmodes.exe*
mapmodes.def	Module definition file for *mapmodes.exe*
mapmodes.h	Include file for *mapmodes.exe*
mapmodes.ico	Icon file for *mapmodes.exe*
mapmodes.rc	Resource script file for *mapmodes.exe*
motion	**Make** file for *motion.exe*
motion.c	Source file for *motion.exe*
motion.def	Module definition file for *motion.exe*
motion.h	Include file for *motion.exe*
motion.ico	Icon file for *motion.exe*
motion.lnk	Link file for *motion.exe*
motion.rc	Resource script file for *motion.exe*
motion1.c	Source file for *motion.exe*
motion2.c	Source file for *motion.exe*
motioni.c	Source file for *motion.exe*
motlib1.c	Source file for *motlib1.exe*, a user-defined library
motlib1.def	Module definition file for *motlib1.exe*
motlib2.c	Source file for *motlib2.exe*, a user-defined library
motlib2.def	Module definition file for *motlib2.exe*
print.c	Source file for *sample.exe*
sample	**Make** file for *sample.exe*
sample.c	Source file for *sample.exe*

sample.def	Module definition file for *sample.exe*
sample.h	Include file for *sample.exe*
sample.ico	Icon file for *sample.exe*
sample.lnk	Link file for *sample.exe*
sample.rc	Resource script file for *sample.exe*
shapes	**Make** file for *shapes.exe*
shapes.c	Source file for *shapes.exe*
shapes.def	Module definition file for *shapes.exe*
shapes.doc	Document file for *shapes.exe*
shapes.h	Include file for *shapes.exe*
shapes.ico	Icon file for *shapes.exe*
shapes.rc	Resource script file for *shapes.exe*
tempinit.c	Source file for *template.exe*'s initialization
template	**Make** file for *template.exe*
template.def	Module definition file for *template.exe*
template.h	Include file for *template.exe*
template.ico	Icon file for *template.exe*
template.rc	Resource script file for *template.exe*
tempnres.c	Source file for *template.exe*'s non-resident segment
tempres.c	Source file for *template.exe*'s resident segment
tools.asm	Source file for *sample.exe*
track	**Make** file for *track.exe*
track.c	Source file for *track.exe*
track.def	Module definition file for *track.exe*
track.doc	Document file for *track.exe*
track.h	Include file for *track.exe*
track.ico	Icon file for *track.exe*
track.rc	Resource script file for *track.exe*

type	**Make** file for *type.exe*
type.c	Source file for *type.exe*
type.def	Module definition file for *type.exe*
type.doc	Document file for *type.exe*
type.h	Include file for *type.exe*
type.ico	Icon file for *type.exe*
type.rc	Resource script file for *type.exe*

A.1.4 Windows Sample Application: Cardfile

File	Purpose
asmsubs.asm	Assembly source file for *cardfile.exe*
cardfile	**Make** file for *cardfile.exe*
cardfile.def	Module definition file for *cardfile.exe*
cardfile.h	Include file for *cardfile.exe*
cardfile.ico	Icon file for *cardfile.exe*
cardfile.lnk	Link file for *cardfile.exe*
cardfile.rc	Resource script file for *cardfile.exe*
cfbitmap.c	C source file for *cardfile.exe*
cfcard.c	C source file for *cardfile.exe*
cfclip.c	C source file for *cardfile.exe*
cfdata.c	C source file for *cardfile.exe*
cfdb.c	C source file for *cardfile.exe*
cfdial.c	C source file for *cardfile.exe*
cfdos.asm	Assembly source file for *cardfile.exe*
cffile.c	C source file for *cardfile.exe*
cffind.c	C source file for *cardfile.exe*
cfinput.c	C source file for *cardfile.exe*
cfmain.c	C source file for *cardfile.exe*

cfnew.c	C source file for *cardfile.exe*
cfopen.c	C source file for *cardfile.exe*
cfpaint.c	C source file for *cardfile.exe*
cfprint.c	C source file for *cardfile.exe*
cfres.c	C source file for *cardfile.exe*
cfscroll.c	C source file for *cardfile.exe*
cftext.c	C source file for *cardfile.exe*
declare.h	Include file for *cardfile.exe*
dlgopen.c	C source file for open dialog box

A.1.5 Windows Sample Application: Pascal Language

File	Purpose
muzzle	**Make** file for *muzzle.exe*
muzzle.cur	Cursor file for *muzzle.exe*
muzzle.def	Module definition file for *muzzle.exe*
muzzle.ico	Icon file for *muzzle.exe*
muzzle.inc	Include file for *muzzle.exe*
muzzle.pas	Source file for *muzzle.exe*
muzzle.rc	Resource script for *muzzle.exe*

A.1.6 Windows Debugging Files

File	Purpose
gdi.exe	Debugging executable file for GDI library
gdi.sym	Symbolic debugging file for GDI functions
kernel.exe	Debugging executable file for kernel library
kernel.sym	Symbolic debugging file for kernel functions

user.exe	Debugging executable file for user library
user.sym	Symbolic debugging file for user functions

A.1.7 C Library and Include Files

File	Purpose
clibw.lib	Standard Windows library (compact model)
cwinlibc.lib	Startup library for Windows libraries (compact model)
llibw.lib	Standard Windows library (large model)
lwinlibc.lib	Startup library for Windows libraries (large model)
mlibw.lib	Standard Windows library (medium model)
mwinlibc.lib	Startup library for Windows libraries (medium model)
slibw.lib	Standard Windows library (small model)
swinlibc.lib	Startup library for Windows libraries (small model)
windows.h	Windows include file for C-language applications

A.1.8 Pascal Library and Include Files

File	Purpose
pascal.lib	Pascal library for Windows applications
paslibw.lib	Windows library for applications in the Pascal language
windows.inc	Windows include file for Pascal language applications
winnames.inc	Windows name list file for the *paslibw.lib* interface

A.1.9 Assembly-Language Files

File	Purpose
cmacros.inc	Macro assembler include file for C macro definitions. These macros can be used to create C- or Pascal-compatible object files.

Appendix B
Diagnostic Messages

The debugging version of Microsoft Windows generates diagnostic messages whenever it encounters an error that would otherwise cause the system to fail. Each diagnostic message has a unique number that identifies the cause of the message and potential failure. Table B.1 lists the diagnostic message numbers and explains the meaning of each message.

Table B.1

Diagnostic Messages

Number (Hex)	Description
0001	Insufficient memory for allocation
0002	Error reallocating memory
0003	Memory cannot be freed
0004	Memory cannot be locked
0005	Memory cannot be unlocked
0007	Window handle not valid
0008	Cached display contexts are busy
0010	Clipboard already open
0013	Mouse module not valid
0014	Display module not valid
0015	Unlocked data segment should be locked
0016	Invalid lock on system queue
0100	Local memory errors
0140	Local heap is busy
0180	Invalid local handle
01C0	**LocalLock** count overflow
01F0	**LocalUnlock** count underflow
0200	Global memory errors
0240	Critical section problems
0280	Invalid global handle
02C0	**GlobalLock** count overflow
02F0	**GlobalUnlock** count underflow
0300	Task schedule errors
0301	Invalid task ID
0302	Invalid exit system call
0303	Invalid **bp** register chain
0400	Dynamic loader/linker errors
0401	Error during boot process

Table B.1 *(continued)*

Number (Hex)	Description
0402	Error loading a module
0403	Invalid ordinal reference
0404	Invalid entry name reference
0405	Invalid start procedure
0406	Invalid module handle
0407	Invalid relocation record
0408	Error saving forward reference
0409	Error reading segment contents
0410	Error reading segment contents
0411	Insert disk for specified file
0412	Error reading non-resident table
04FF	INT 3F handler unable to load segment
0500	Resource manager/user profile errors
0501	Missing resource table
0502	Bad resource type
0503	Bad resource name
0504	Bad resource file
0505	Error reading resource
0600	Atom manager errors
0700	Input/output package errors

Appendix C
C Run-time Functions

Table C.1 lists all C run-time functions and indicates whether the function code assumes that the **ds** and **ss** registers are equal. Only C run-time functions that assume **ds** and **ss** are not equal can be used in Windows libraries.

Table C.1
C Run-time Functions

Function	ds !=ss	Function	ds !=ss
_clear87	yes	**bessel**	yes
_control87	yes	**bsearch**	yes
_exit	no	**cabs**	yes
_expand	yes	**calloc**	yes
_ffree	yes	**ceil**	yes
_fmalloc	yes	**cgets**	yes
_fmsize	yes	**chdir**	yes
_fpreset	yes	**chmod**	yes
_freect	yes	**chsize**	no
_memavl	yes	**clearerr**	yes
_msize	yes	**close**	yes
_nfree	yes	**cos**	yes
_nmalloc	yes	**cosh**	yes
_nmsize	yes	**cprintf**	no
_status87	yes	**cputs**	yes
abort	no	**creat**	no
abs	yes	**cscanf**	no
access	yes	**ctime**	yes
acos	yes	**dieeetomsbin**	yes
alloca	yes	**difftime**	yes
asctime	yes	**dmsbintoiees**	yes
asin	yes	**dosexterr**	yes
atan	yes	**dup**	yes
atan2	yes	**dup2**	yes
atof	yes	**ecvt**	yes
atoi	yes	**eof**	yes
atol	yes	**execl**	no
bdos	yes	**execle**	no

Table C.1 *(continued)*

Function	ds !=ss	Function	ds !=ss
execlp	no	**getw**	yes
execlpe	no	**gmtime**	yes
execv	no	**halloc**	yes
execve	no	**hfree**	yes
execvp	no	**hypot**	yes
execvpe	no	**inp**	yes
exit	no	**int86**	yes
exp	yes	**int86x**	yes
fabs	yes	**intdos**	yes
fclose	yes	**intdosx**	yes
fcloseall	yes	**isatty**	yes
fcvt	yes	**itoa**	yes
fdopen	yes	**kbhit**	yes
fgetc	yes	**labs**	yes
fgetchar	yes	**ldexp**	yes
fgets	yes	**lfind**	yes
fieeetomsbin	yes	**localtime**	yes
filelength	yes	**locking**	yes
fllush	yes	**log**	yes
floor	yes	**log10**	yes
flushall	yes	**longjmp**	yes
fmod	yes	**lsearch**	yes
fmsbintoieee	yes	**lseek**	yes
fopen	yes	**ltoa**	yes
fprintf	no	**malloc**	yes
fputc	yes	**matherr**	yes
fputchar	yes	**memccpy**	yes
fputs	yes	**memchr**	yes
fread	yes	**memcmp**	yes
free	yes	**memcpy**	yes
freopen	yes	**memicmp**	yes
frexp	yes	**memset**	yes
fscanf	no	**mkdir**	yes
fseek	yes	**mktemp**	yes
fstat	yes	**modf**	yes
ftell	yes	**movedata**	yes
ftime	yes	**onexit**	yes
fwrite	yes	**open**	yes
gcvt	yes	**outp**	yes
getch	yes	**perror**	yes
getche	yes	**pow**	yes
getcwd	yes	**printf**	no
getenv	yes	**putch**	yes
getpid	yes	**putenv**	yes
gets	yes	**puts**	yes

Table C.1 *(continued)*

Function	ds !=ss	Function	ds !=ss
putw	yes	**strerror**	yes
qsort	yes	**stricmp**	yes
rand	yes	**strlen**	yes
read	yes	**strlwr**	yes
realloc	yes	**strncat**	yes
remove	yes	**strncmp**	yes
rename	yes	**strncpy**	yes
rmdir	yes	**strnicmp**	yes
rmtmp	yes	**strnset**	yes
sbrk	yes	**strpbrk**	yes
scanf	no	**strrchr**	yes
segread	yes	**strrev**	yes
setbuf	yes	**strset**	yes
setjmp	yes	**strspn**	yes
setmode	yes	**strstr**	yes
setvbuf	yes	**strtod**	yes
signal	yes	**strtok**	yes
sin	yes	**strtol**	yes
sinh	yes	**strupr**	yes
sopen	yes	**swab**	yes
spawnl	no	**system**	yes
spawnle	no	**tan**	yes
spawnlp	no	**tanh**	yes
spawnlpe	no	**tell**	yes
spawnv	no	**tempnam**	yes
spawnve	no	**time**	yes
spawnvp	no	**tmpfile**	no
spawnvpe	no	**tmpnam**	yes
sprintf	no	**tolower**	yes
sqrt	yes	**toupper**	yes
srand	yes	**tzset**	yes
sscanf	no	**ultoa**	yes
stackavail	yes	**umask**	yes
stat	yes	**ungetc**	yes
strcat	yes	**ungetch**	yes
strchr	yes	**unlink**	yes
strcmp	yes	**utime**	yes
strcmpi	yes	**vfprintf**	no
strcpy	yes	**vprintf**	no
strcspn	yes	**vsprintf**	no
strdup	yes	**write**	yes

Index