Microsoft® Windows

Software Development Kit

Programming Tools

Version 2.0

Microsoft Corporation

Document No. 050051051-200-I01-1087
Part No. 00476

Contents

Figures

Tables

Chapter 1
Introduction

1.1 About This Guide

Microsoft Windows Programming Tools is divided into two parts: "DOS Tools" and "Application Development Tools." The "DOS Tools" 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 tools use command lines and parameters for input. "DOS Tools" includes descriptions of the following programming tools:

Program	Description
cl	Compiles C-language 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.
cmacros	Creates assembly-language Windows applications that are compatible with C and Pascal Windows applications.

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

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

1.2 Notational Conventions

The following notational conventions are used throughout this manual:

Convention	Description
bold	Bold type is used to indicate commands, functions, statements, directives, options, macros, registers, data types, structures, and keywords, which must be used exactly as shown. In the following example, **–c, –AS, –Gsw, –Os**, and **–Zdp** are options of the **cl** command: **cl –c –AS –Gsw –Os –Zdp** *test.c*
italic	Italic is used for placeholders—descriptive names that represent parameters, fields, or variables that the programmer must supply. In the following example, *text* and *optionlist* are fields which must be supplied with specific data that will be passed to the **POPUP** statement: **POPUP** *text optionlist*
⟦**double brackets**⟧	Double brackets enclose optional fields or parameters in syntax statements. In the following example, *option* and *executable-file* are optional parameters of the **rc** command: **rc** ⟦*option*⟧ *filename* ⟦*executable-file*⟧
ellipses...	Ellipses following an item indicate that more items that have the same form may appear. Ellipses may be horizontal or vertical. In the following example, the ellipses at the end indicate that more than one *breakaddress* may be specified for the **g** command: **g** ⟦=*startaddress*⟧ ⟦*breakaddress*⟧... In the following example, the ellipses between the lines indicate that intervening program lines occur but are not shown: *acctablename* **ACCELERATORS** **BEGIN** *event, idvalue,* ⟦*type*⟧ ⟦NOINVERT⟧ ⟦SHIFT⟧ ⟦CONTROL⟧ . . . **END**

CONTROL+C	Keynames appear in small capital letters. A plus sign (+) is used to indicate key combinations. A key combination requires that you press and hold down the first key and then press the next key(s).

DOS Tools

Chapter 2
Cl, Pascal, and the Macro Assembler

2.1 Introduction

A complete application for the Microsoft Windows presentation manager begins with the application's program and resource source files. You can write Windows applications in C, Pascal, or assembly language. You can add 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 that lists (or defines) 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. Small-model applications are usually small applications that cannot be divided easily into separate code segments. Use the **–AS** option, if desired. (The option is not really 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 that the following statement must be in the module-definition file of any compact- or large-model application:

```
DATA FIXED
```

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

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 callback 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.

2.2.4 Callback Functions

Callback functions are functions in an application that Windows calls in order 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, you must use the **–Gw** option with the **cl** command when you compile any application source files containing functions that can be called with 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 Application

You compile Windows application source files by using the **cl** command. Since the object files generated by **cl** must be linked with the Windows linker (**link4**), you should use the **–c** option to prevent **cl** from attempting to create a file that is not executable under Windows. You should give other options, such as those specifying programming model, packed structures, and Windows prolog and epilog, 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 that 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, so you should use the –**Os** option to direct the compiler to optimize for code size instead of speed. If you don't want optimizing, use the –**Od** option.

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 debugging 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. Which C-language libraries you link with depends on your application's programming model. For example, a small-model application must be linked with the small-model library *slibw.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, "Windows Linker: Link4." You need an additional library, *win87em.lib*, when you use the coprocessor/emulator floating-point option.

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 when you supply 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 *win200.bin.* This means that the **__argv**[0] value always points to a string that contains *win200.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

C-language Windows applications that use floating-point variables must specify a particular floating-point option during compilation and must give additional library names on the **link4** command line. The examples in the following sections apply to small applications (**–AS**), but you can adapt them to the other programming models by changing the initial "s" in certain library names to "c", "m", or "l", as appropriate. See the *Microsoft C Compiler User's Guide* for more information about the different floating-point libraries and options. (Note that the libraries *win87em.lib* and *win87em.exe* take the place of *em.lib* and *87.lib*, which are used in the DOS environment.)

2.2.14.1 Coprocessor/Emulator Math Option

You can choose the coprocessor/emulator math option by specifying **–FPi** and **–FPc** on the compiler command line and by linking with *slibfw.lib* and *win87em.lib* on the **link4** command line.

Example

```
cl -c -FPi -AS -Gsw -Os -Zpe sample.c
link4 sample,/align:16,/map,win87em slibw,sample.def;
```

If you link the application with *win87em.lib*, you can run the application either with or without a numeric coprocessor (8087/80827/80387). If there is a coprocessor present, the application uses it; if there is no coprocessor present, the application's floating-point emulator simulates the operation of the coprocessor. To use the coprocessor/emulator option, you must have the dynamic-link library *win87em.exe* either in the current directory or along the executable search path when the application is loaded. (The application developer should distribute the *win87em.exe* library with applications that use it.) This is a fast option when there is a coprocessor available.

2.2.14.2 Alternate Math Option

You can choose the alternate math option by specifying **–FPa** on the compiler command line and by linking with *slibw.lib* on the **link4** command line.

Example

```
cl -c -FPa -AS -Gsw -Os -Zpe sample.c
link4 sample,/align:16,/map,slibw,sample.def;
```

This option does not use the coprocessor to perform floating-point operations. It is faster than the emulator when no coprocessor is present, but slower than the coprocessor math option when the coprocessor is present.

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 Programmer's Learning 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 values of the **ds** and **ss** registers are not equal when the library is executed. Since the C compiler generates code that assumes that 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 B, "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 not declare 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 by 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.

For information on using Pascal-language Windows applications, see the *Microsoft Pascal Compiler Version 4.0 Update.*

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 Chapter 7, "Assembly-Language Macros."
2. Define the programming model, setting one of the options **memS**, **memM**, **memC**, or **memL** to 1. This option must be set before you specify the statement that includes the *cmacros.inc* file.
3. Set the calling convention to Pascal by setting the **?PLM** option to 1. This option must be set before you specify the statement that includes 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 that 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 you specify the statement that includes the *cmacros.inc* file. This option is required only for callback functions (or for exported functions in Windows libraries).
6. Make sure that your callback functions are declared

   ```
   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 presentation manager 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. The resource script file defines the names and attributes of the resources to be added to the application's executable file. The file consists of one or more "resource statements" that define the resource type and original 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's name, type, and details.

The file also contains one or more directives, which are special statements that define actions to be performed 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.

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

The following is a list of the resource statements:

Type of Statement	Statement
Single-line statements	**CURSOR** **ICON** **BITMAP** **FONT**
User-defined resources	User-supplied
Multiple-line statements	**RCDATA** **STRINGTABLE** **ACCELERATORS** **MENU** **DIALOG**

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

Example

```
#include "shapes.h"
shapes  cursor  shapes.cur
shapes  icon    shapes.ico
shapes  menu
begin
    pop-up "&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 be performed 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 you use 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* field specifies an ASCII string, enclosed in double quotation marks, that identifies 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* field 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
```

In this example, 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 ⟦*option*⟧ *filename* ⟦*executable-file*⟧

The *option* parameter allows the selection of one of the following options:

Option	Description
–r	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.
–LIM32	Sets **rc** to compile an application that uses expanded memory according to the Lotus Intel Microsoft Expanded Memory Specification, Version 3.2.
–multinst	Sets **rc** to compile an application that uses expanded memory so that multiple instances of the application will use different EMS banks.

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

The *executable-file* parameter specifies 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.

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
```

3.4 Single-Line Statements

The single-line statements define resources that are contained in a single file, such as cursors, icons, and fonts. The statements associate the filename of the resource with an identifying name or number. The resource is added to the executable file when the application is created, and can be extracted during execution by referring to the name or number.

The following is the general form for all single-line statements:

nameID resource-type ⟦*load-option*⟧ ⟦*mem-option*⟧ *filename*

The *nameID* field specifies either a unique name or an integer value identifying the resource. For a font resource, *nameID* must be a number; it cannot be a name.

The *resource-type* field specifies one of the following keywords, which identify the type of resource to be loaded:

Keyword	Resource Type
CURSOR	Specifies a bitmap that defines the shape of the mouse cursor on the display screen.
ICON	Specifies a bitmap that defines the shape of the icon to be used for a given application.
BITMAP	Specifies a custom bitmap that an application is going to use in its screen display or as an item in a menu.
FONT	Specifies a file that contains a font.

The *load-option* field takes an optional keyword that specifies when the resource is to be loaded. It must be one of the following:

PRELOAD	Resource is loaded immediately.
LOADONCALL	Resource is loaded when called.

The default is **LOADONCALL.**

The optional *mem-option* field takes the following keyword or keywords, which specify whether the resource is fixed or movable and whether it is discardable:

FIXED	Resource remains at a fixed memory location.
MOVEABLE	Resource can be moved if necessary to compact memory.
DISCARDABLE	Resource can be discarded if no longer needed.

The default is **MOVEABLE** and **DISCARDABLE** for cursor, icon, and font resources. The default for bitmap resources is **MOVEABLE.**

The *filename* field takes an ASCII string that specifies the DOS filename of the file containing the resource. A full pathname must be given if the file is not in the current working directory.

Examples

```
cursor CURSOR point.cur
cursor CURSOR DISCARDABLE point.cur
10      CURSOR custom.cur

desk    ICON desk.ico
desk    ICON DISCARDABLE desk.ico
11      ICON custom.ico

disk    BITMAP disk.bmp
disk    BITMAP DISCARDABLE disk.bmp
12      BITMAP custom.bmp

5 FONT  CMROMAN.FON
```

3.5 User-Defined Resources

An application can also define its own resource. The resource can be any data that the application intends to use. A user-defined resource statement has the following form:

nameID typeID ⟦*load-option*⟧ ⟦*mem-option*⟧ *filename*

The *nameID* field specifies either a unique name or an integer value identifying the resource.

The *typeID* field specifies either a unique name or an integer value identifying the resource type. If a number is given, it must be greater than 255. The numbers 1 through 255 are reserved for existing and future predefined resource types.

The *load-option* field takes an optional keyword that specifies when the resource is to be loaded. It must be one of the following:

PRELOAD	Resource is loaded immediately.
LOADONCALL	Resource is loaded when called.

The default is **LOADONCALL.**

The optional *mem-option* field takes the following keyword or keywords, which specify whether the resource is fixed or movable and whether it is discardable:

FIXED	Resource remains at a fixed memory location.
MOVEABLE	Resource can be moved if necessary to compact memory.
DISCARDABLE	Resource can be discarded if no longer needed.

The default is **MOVEABLE.**

The *filename* field takes an ASCII string specifying the DOS filename of the file containing the resource. A full pathname must be given if the file is not in the current working directory.

Example

```
array   MYRES  data.res
14      300     custom.res
```

3.6 RCDATA Statement

Syntax

RCDATA ⟦*load-option*⟧ ⟦*mem-option*⟧
BEGIN
raw-data
END

The **RCDATA** statement defines one or more more raw data resources for an application. Raw data resources permit the inclusion of binary data directly into the executable file.

The *load-option* field takes an optional keyword that specifies when the resource is to be loaded. It must be one of the following:

PRELOAD	Resource is loaded immediately.
LOADONCALL	Resource is loaded when called.

The default is **LOADONCALL.**

The optional *mem-option* field takes the following keyword or keywords, which specify whether the resource is fixed or movable and whether it is discardable:

FIXED	Resource remains at a fixed memory location.
MOVEABLE	Resource can be moved if necessary to compact memory.
DISCARDABLE	Resource can be discarded if no longer needed.

The default is **MOVEABLE** and **DISCARDABLE.**

The *raw-data* field specifies one or more integers and strings stated in standard C-language format. Integers are given in decimal, octal, or hexadecimal format.

Example

```
resname RCDATA
BEGIN
   "Here is a data string\0",  /* A string. Note: not null-terminated */
   1024,                        /* int                                 */
   0x029a,                      /* hex int                             */
   0o733,                       /* octal int                           */
   "\07"                       /* octal byte                          */
END
```

3.7 STRINGTABLE Statement

Syntax

STRINGTABLE ⟦*load-option*⟧ ⟦*mem-option*⟧
BEGIN
string-definitions
END

The **STRINGTABLE** statement defines one or more more string resources for an application. String resources are simply null-terminated ASCII strings that can be loaded when needed from the executable file, using the **LoadString** function.

The *load-option* field takes an optional keyword that specifies when the resource is to be loaded. It must be one of the following:

PRELOAD	Resource is loaded immediately.
LOADONCALL	Resource is loaded when called.

The default is **LOADONCALL**.

The optional *mem-option* field takes the following keyword or keywords, which specify whether the resource is fixed or movable and whether it is discardable:

FIXED	Resource remains at a fixed memory location.
MOVEABLE	Resource can be moved if necessary to compact memory.
DISCARDABLE	Resource can be discarded if no longer needed.

The default is **MOVEABLE** and **DISCARDABLE**.

The *string-definitions* field specifies one or more ASCII strings, enclosed in double quotation marks and preceded by an identifier. The identifier must be an integer.

Example

```
#define IDS_HELLO    1
#define IDS_GOODBYE  2

STRINGTABLE
BEGIN
    IDS_HELLO,   "Hello"
    IDS_GOODBYE, "Goodbye"
END
```

3.8 ACCELERATORS Statement

Syntax

acctablename **ACCELERATORS**
BEGIN
event, idvalue, ⟦*type*⟧ ⟦**NOINVERT**⟧ ⟦**SHIFT**⟧ ⟦**CONTROL**⟧
.
.
.
END

The **ACCELERATORS** statement defines one or more accelerators for an application. An accelerator is a keystroke defined by the application to give the user a quick way to perform a task. The **TranslateAccelerator** function is used to translate accelerator messages from the application queue into WM_COMMAND or WM_SYSCOMMAND messages.

The *acctablename* field specifies the name of the accelerator table.

The *event* field specifies the keystroke to be used as an accelerator. It can be any one of the following:

Character	Description
"*char*"	A single character enclosed in double quotes. The character can be preceded by a caret (^), meaning that the character is a control character.
ASCII character	An integer value representing an ASCII character. The *type* field must be **ASCII**.
Virtual key character	An integer value representing a virtual key. The *type* field must be **VIRTKEY**.

The *idvalue* field specifies an integer value that identifies the accelerator.

The *type* field is required only when *event* is an ASCII character or a virtual key character. The *type* field specifies either **ASCII** or **VIRTKEY**; the integer value of *event* is interpreted accordingly.

The **NOINVERT** option, if given, means that no top-level menu item is highlighted when the accelerator is used. This is useful when defining accelerators for actions such as scrolling that do not correspond to a menu item. If **NOINVERT** is omitted, a top-level menu item will be highlighted (if possible) when the accelerator is used.

The **SHIFT** option, if given, causes the accelerator to be activated only if the SHIFT key is down.

The **CONTROL** option, if given, defines the character as a control character (the accelerator is only activated if the CONTROL key is down). This has the same effect as using a caret (^) before the accelerator character in the *event* field.

Examples

```
MainAcc ACCELERATORS
BEGIN
        "^S", ID_SAVE, NOINVERT
        VK_UP, 6, VIRTKEY, NOINVERT
        7, ID_BELL, ASCII
        "^g", ID_BELL
        "G", ID_BELL, CONTROL
END
```

Note that the last three definitions are equivalent.

3.9 MENU Statement

Syntax

menuID **MENU** [[*load-option*]] [[*mem-option*]]
BEGIN
item-definitions
END

The **MENU** statement defines the contents of a menu resource. A menu resource is a collection of information that defines the appearance and function of an application menu. A menu is a special input tool that lets a user select commands from a list of command names.

The *menuID* field specifies a name or number used to identify the menu resource.

The *load-option* field takes an optional keyword that specifies when the resource is to be loaded. It must be one of the following:

PRELOAD	Resource is loaded immediately.
LOADONCALL	Resource is loaded when called.

The default is **LOADONCALL.**

The optional *mem-option* field takes the following keyword or keywords, which specify whether the resource is fixed or movable and whether it is discardable:

FIXED	Resource remains at a fixed memory location.
MOVEABLE	Resource can be moved if necessary to compact memory.
DISCARDABLE	Resource can be discarded if no longer needed.

The *item-definition* field specifies special resource statements which define the items in the menu.

Example

The following is an example of a complete **MENU** statement.

```
sample MENU
BEGIN
        Menuitem "&Alpha", 100
        POPUP "&Beta"
        BEGIN
                Menuitem "&Item 1", 200
                MENUITEM "I&tem 2", 201, CHECKED
        END
END
```

3.9.1 Item-Definition Statements

The **MENUITEM** and **POPUP** statements are used in the *item-definition* section of a **MENU** statement to define the names and attributes of the actual menu items. Any number of statements can be given; each defines a unique item. The order of the statements defines the order of the menu items.

The **MENUITEM** and **POPUP** statements can only be used within an *item-definition* section of a **MENU** statement.

3.9.1.1 MENUITEM Statement

Syntax

MENUITEM *text, result, optionlist*

This optional statement defines a menu item.

The *text* field takes an ASCII string, enclosed in double quotation marks, which specifies the name of the menu item.

The string can contain the escape characters **\t** and **\a.** The **\t** character inserts a tab in the string and is used to align text in columns. Tab characters should be used only in pop-up menus, not in menu bars. (See Section 3.9.1.2 for information on pop-up menus.) The **\a** character sets all text that follows it flush right.

To insert a double quotation mark (") in the string, use two double quotation marks ("").

To add a mnemonic to the text string, place the ampersand (&) ahead of the letter that will be the mnemonic. This will cause the letter to appear underlined in the control and to function as the mnemonic. To use the ampersand as a character in a string, insert two ampersands (&&).

The *result* field takes an integer value that specifies the result generated when the user selects the menu item. Menu-item results are always integers; when the user clicks the menu-item name, the result is sent to the window that owns the menu.

The *optionlist* field takes one or more predefined menu options, separated by commas or spaces, that specify the appearance of the menu item. The menu options are as follows:

Option	Description
MENUBREAK	Item is immediately preceded by a new line.
CHECKED	Item has a checkmark next to it.
INACTIVE	Item name is displayed, but cannot be selected.
GRAYED	Item name is initially inactive and appears on the menu in gray or a lightened shade of the menu-text color.
HELP	Item is flush right on the menu bar, with a vertical separator to its left. The Help item cannot be selected from the keyboard.

The **INACTIVE** and **GRAYED** options cannot be used together.

Examples

```
MENUITEM  "&Alpha", 1, CHECKED, GRAYED
MENUITEM  "&Beta", 2
```

3.9.1.2 POPUP Statement

Syntax

POPUP *text, optionlist*
BEGIN
item-definitions
END

This statement marks the beginning of a pop-up menu definition. A pop-up menu (which is also known as a drop-down menu) is a special menu item that displays a sublist of menu items when it is selected.

The *text* field takes an ASCII string, enclosed in double quotation marks, which specifies the name of the pop-up.

The *optionlist* field takes one or more predefined menu options that specify the appearance of the menu item. The menu options are as follows:

Option	Description
MENUBREAK	Item is placed in a new column.
MENUBARBREAK	Item is placed in a new column. The old and new columns are separated with a bar.
CHECKED	Item has a checkmark next to it.
INACTIVE	Item name is displayed, but cannot be selected.
GRAYED	Item name is initially inactive and appears on the menu in gray or a lightened shade of the menu-text color.

The options can be combined using the bitwise OR operator. The **INACTIVE** and **GRAYED** options cannot be used together.

The *item-definitions* field can specify any number of **MENUITEM** statements. **POPUP** statements in a pop-up menu are not allowed.

Example

```
chem MENU
BEGIN

POPUP "&Elements"
BEGIN
        Menuitem "&Oxygen", 200
        Menuitem "&Carbon", 201, CHECKED
        Menuitem "&Hydrogen", 202
END

POPUP "&Compounds", CHECKED
BEGIN
        Menuitem "&Glucose", 301
        Menuitem "&Sucrose", 302, CHECKED
        Menuitem "&Lactose", 303, MENUBREAK
        Menuitem "&Fructose", 304
END

END
```

3.9.1.3 MENUITEM SEPARATOR Statement

Syntax

MENUITEM SEPARATOR

This special form of the **MENUITEM** statement creates an inactive menu item that serves as a dividing bar between two active menu items. In pop-up menus, the bar is horizontal. In a menu bar, the dividing bar is vertical.

Example

```
MENUITEM "&Roman", 206
MENUITEM SEPARATOR
MENUITEM "&20 Point", 301
```

3.10 DIALOG Statement

The **DIALOG** statement defines a template that can be used by an application to create dialog boxes.

Syntax

nameID **DIALOG** ⟦*load-option*⟧ ⟦*mem-option*⟧ *x, y, width, height*
⟦*option-statements*⟧
BEGIN
control-statements
END

This statement marks the beginning of a **DIALOG** template. It defines the name of the dialog box, the memory and load options, the box's starting location on the display screen, and its width and height.

The *nameID* field specifies either a unique name or an integer value that identifies the resource.

The *load-option* field takes an optional keyword that specifies when the resource is to be loaded. It must be one of the following:

PRELOAD	Resource is loaded immediately.
LOADONCALL	Resource is loaded when called.

The default is **LOADONCALL.**

The optional *mem-option* field takes the following keyword or keywords, which specify whether the resource is fixed or movable and whether it is discardable:

FIXED	Resource remains at a fixed memory location.
MOVEABLE	Resource can be moved if necessary to compact memory.
DISCARDABLE	Resource can be discarded if no longer needed.

The default is **MOVEABLE.**

The *x* and *y* fields take integer values that specify the *x* and *y* coordinates of the upper-left corner of the dialog box. The exact meaning of the coordinates depends on the style defined by the **STYLE** statement. For child-style dialog boxes, the coordinates are relative to the origin of the parent window, unless the dialog box has the style DS_ABSALIGN; in that case the coordinates are relative to the origin of the display screen.

The *width* and *height* fields take integer values that specify the width and height of the box. The width units are 1/4 of the width of a character; the height units are 1/8 of the height of a character.

The option and control statements are described in the following sections.

Example

```
errmess DIALOG  10, 10, 300, 200
```

The following is a complete example of a **DIALOG** statement.

```
#include "windows.h"

errmess DIALOG  10, 10, 300, 110
STYLE WS_POPUP|WS_BORDER
CAPTION "Error!"
BEGIN
        CTEXT "Select One:", 1, 10, 10, 280, 12
        RADIOBUTTON "&Retry", 2, 75, 30, 60, 12
        RADIOBUTTON "&Abort", 3, 75, 50, 60, 12
        RADIOBUTTON "&Ignore", 4, 75, 80, 60, 12
END
```

Comments

Do not use the WS_CHILD style with a modal dialog box. The **DialogBox** function always disables the parent/owner of the newly-created dialog box. When a parent window is disabled, its child windows are implicitly disabled. Since the parent window of the child-style dialog box is disabled, so is the child-style dialog box itself.

If a dialog box has the DS_ABSALIGN style, the dialog coordinates for its upper-left corner are relative to the screen origin instead of to the upper-left corner of the parent window. You would typically use this style when you want the dialog box to start in a specific part of the display no matter where the parent window may be on the screen.

The name **DIALOG** can also be used as the class-name parameter to the **CreateWindow** function in order to create a window with dialog-box attributes.

3.10.1 Dialog Option Statements

The dialog option statements, given in the *option-statements* section of the **DIALOG** statement, define special attributes of the dialog box, such as its style, caption, and menu. The option statements are optional. If there are no option statements, the dialog box is given a default attribute. Dialog option statements include the following:

- **STYLE**
- **CAPTION**
- **MENU**
- **CLASS**

The option statements are discussed individually in the following sections.

3.10.1.1 STYLE Statement

Syntax

STYLE ***style***

This optional statement defines the window style of the dialog box. The window style specifies whether the box is a pop-up or a child window. The default style has the following attributes:

WS_POPUP
WS_BORDER
WS_SYSMENU

The *style* field takes an integer value or predefined name that specifies the window style. It can be any of the window styles defined in the following Table 3.1.

The style DS_SYSMODAL can also be used to create a system modal dialog box.

Comments

If the predefined names are used, the **#include** directive must be used so that the *windows.h* file will be included in the resource script.

Table 3.1

Window Styles

Style	Meaning
WS_GROUP	Specifies the first control of a group of controls in which the user can move from one control to the next by using the cursor keys. All controls defined with the WS_GROUP style after the first control belong to the same group. The next control with the WS_GROUP style ends the style group and starts the next group (i.e., one group ends where the next begins).
WS_TABSTOP	Specifies one of any number of controls through which the user can move by using the TAB key. The TAB key moves the user to the next control specified by the WS_TABSTOP style.
WS_POPUP	Creates a pop-up window. Cannot be used with WS_CHILD.
WS_CHILD	Creates a child window. Cannot be used with WS_POPUP.
WS_ICONIC	Creates a window that is initially iconic. For use with WS_OVERLAPPED only.
WS_OVERLAPPED	Creates an overlapping window.
WS_OVERLAPPEDWINDOW	Creates an overlapped window having the styles WS_OVERLAPPED, WS_CAPTION, WS_SYSMENU, and WS_SIZEBOX.
WS_MINIMIZE	Creates a window of minimum size.
WS_MAXIMIZE	Creates a window of maximum size.
WS_BORDER	Creates a window that has a border.
WS_CAPTION	Creates a window that has a title bar (implies WS_BORDER).
WS_DLGFRAME	Creates a window with a double border but no title.
WS_SYSMENU	Creates a window that has a system-menu box in its title bar. Used only for windows with title bars. If used with a child window, this style creates a Close box instead of a system-menu box.
WS_MINIMIZEBOX	Creates a window that has a Minimize box.
WS_MAXIMIZEBOX	Creates a window that has a Maximize box.
WS_SIZEBOX	Creates a window that has a Size box. Used only for windows with a title bar or with vertical and horizontal scroll bars.

Table 3.1 *(continued)*

Style	Meaning
WS_VSCROLL	Creates a window that has a vertical scroll bar.
WS_HSCROLL	Creates a window that has a horizontal scroll bar.
WS_CLIPCHILDREN	Excludes the area occupied by child windows when drawing within the parent window. Used when creating the parent window.
WS_CLIPSIBLINGS	Clips child windows relative to each other; that is, when a particular child window receives a WP_PAINT message, this style clips all other top-level child windows out of the region of the child window to be updated. (If WS_CLIPSIBLINGS is not given and child windows overlap, it is possible, when drawing in the client area of a child window, to draw in the client area of a neighboring child window.) For use with WS_CHILD only.
WS_VISIBLE	Creates a window that is initially visible. This applies to overlapping and pop-up windows. For overlapping windows, the *y* parameter is used as a **ShowWindow** function parameter.
WS_DISABLED	Creates a window that is initially disabled.
WS_POPUPWINDOW	Creates a pop-up window that has the styles WS_POPUP, WS_BORDER, and WS_SYSMENU.
WS_CHILDWINDOW	Creates a child window that has the style WS_CHILD.

3.10.1.2 CAPTION Statement

Syntax

CAPTION *captiontext*

This optional statement defines the dialog box's title. The title appears in the box's caption bar (if it has one).

The default caption is empty.

The *captiontext* field specifies an ASCII character string enclosed in double quotation marks.

Example

```
CAPTION "Error!"
```

3.10.1.3 MENU Statement

Syntax

MENU *menuname*

This optional statement defines the dialog box's menu. If no statement is given, the dialog box has no menu.

The *menuname* field specifies the resource name or number of the menu to be used.

Example

```
MENU errmenu
```

3.10.1.4 CLASS Statement

Syntax

CLASS *class*

This optional statement defines the class of the dialog box. If no statement is given, the predefined dialog class will be used as the default.

The *class* field specifies an integer or a string, enclosed in double quotation marks, that identifies the class of the dialog box.

Example

```
CLASS "myclass"
```

Comments

The **CLASS** statement should be used with special cases, since it overrides the normal processing of a dialog box. The **CLASS** statement converts a dialog box to a window of the specified class; depending on the class, this may give undesirable results. Do not use the predefined control class names with this statement.

3.10.2 Dialog Control Statements

The dialog control statements, given in the *control-statements* section of the **DIALOG** statement, define the attributes of the control windows that appear in the dialog box. A dialog box is empty unless one or more control statements are given. Control statements include the following:

- **LTEXT**
- **RTEXT**
- **CTEXT**
- **CHECKBOX**
- **PUSHBUTTON**
- **LISTBOX**
- **GROUPBOX**
- **DEFPUSHBUTTON**
- **RADIOBUTTON**
- **EDITTEXT**
- **ICON**
- **CONTROL**

The control statements are discussed individually in the following sections. For more information on control classes and styles, see Tables 3.2 and 3.3.

3.10.2.1 LTEXT Statement

Syntax

LTEXT *text, id, x, y, width, height,* [[*style*]]

This statement defines a flush-left text control. It creates a simple rectangle that displays the given text flush-left in the rectangle. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next line.

The *text* field takes an ASCII string that specifies the text to be displayed. The string must be enclosed in double quotation marks. To add a mnemonic to the text string, place the ampersand (&) ahead of the letter that will be the mnemonic. To use the ampersand as a character in a string, insert two ampersands (&&).

The *id* field takes a unique integer value that identifies the control.

The *x* and *y* fields take integer values that specify the *x* and *y* coordinates of the upper-left corner of the control. The coordinates are relative to the origin of the dialog box.

The *width* and *height* fields take integer values that specify the width and height of the control. The width units are 1/4 of the width of a character; the height units are 1/8 of the height of a character.

The optional *style* field contains WS_TABSTOP and/or WS_GROUP styles, which are fully described in Table 3.1. Styles can be combined using the bitwise OR operator.

Comments

The *x*, *y*, *width*, and *height* fields can use addition and subtraction operators (+ and –) for relative positioning. For example, "15 + 6" can be used for the *x* field.

Default style for **LTEXT** is SS_LEFT, WS_GROUP.

Example

```
LTEXT "Enter Name:", 3, 10, 10, 40, 10
```

3.10.2.2 RTEXT Statement

Syntax

RTEXT ***text, id, x, y, width, height,*** ⟦***style***⟧

This statement defines a flush-right text control. It creates a simple rectangle that displays the given text flush-right in the rectangle. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next line.

The *text* field takes an ASCII string that specifies the text to be displayed. The string must be enclosed in double quotation marks. To add a mnemonic to the text string, place the ampersand (&) ahead of the letter that will be the mnemonic. To use the ampersand as a character in a string, insert two ampersands (&&).

The *id* field takes a unique integer value that identifies the control.

The *x* and *y* fields take integer values that specify the *x* and *y* coordinates of the upper-left corner of the control. The coordinates are relative to the origin of the dialog box.

The *width* and *height* fields take integer values that specify the width and height of the control. The width units are 1/4 of the width of a character; the height units are 1/8 of the height of a character.

The optional *style* field contains WS_TABSTOP and/or WS_GROUP styles, which are fully described in Table 3.1. Styles can be combined using the bitwise OR operator.

Comments

The *x*, *y*, *width*, and *height* fields can use addition and subtraction operators (+ and −) for relative positioning. For example, "15 + 6" can be used for the *x* field.

Default style for **RTEXT** is SS_RIGHT, WS_GROUP.

Example

```
RTEXT "Number of Messages", 4, 30, 50, 100, 10
```

3.10.2.3 CTEXT Statement

Syntax

CTEXT *text*, *id*, *x*, *y*, *width*, *height*, [[*style*]]

This statement defines a centered text control. It creates a simple rectangle that displays the given text centered in the rectangle. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next line.

The *text* field takes an ASCII string that specifies the text to be displayed. The string must be enclosed in double quotation marks. To add a mnemonic to the text string, place the ampersand (&) ahead of the letter that will be the mnemonic. To use the ampersand as a character in a string, insert two ampersands (&&).

The *id* field takes a unique integer value that identifies the control.

The *x* and *y* fields take integer values that specify the *x* and *y* coordinates of the upper-left corner of the control. The coordinates are relative to the origin of the dialog box.

The *width* and *height* fields take integer values that specify the width and height of the control. The width units are 1/4 of the width of a character; the height units are 1/8 of the height of a character.

The optional *style* field contains WS_TABSTOP and/or WS_GROUP styles, which are fully described in Table 3.1. Styles can be combined using the bitwise OR operator.

Comments

The *x*, *y*, *width*, and *height* fields can use addition and subtraction operators (+ and −) for relative positioning. For example, "15 + 6" can be used for the *x* field.

Default style for **CTEXT** is SS_CENTER, WS_GROUP.

Example

```
CTEXT "Title", 3, 10, 50, 40, 10
```

3.10.2.4 CHECKBOX Statement

Syntax

CHECKBOX ***text*****,** ***id*****,** ***x*****,** ***y*****,** ***width*****,** ***height*****,** [[*style*]]

This statement defines a check-box control belonging to the BUTTON class. It creates a small rectangle (check box) that is highlighted when clicked. The given text is displayed just to the right of the check box. The control highlights the rectangle when the user clicks the mouse in it, and removes the highlight on the next click.

The *text* field takes an ASCII string that specifies the text to be displayed. The string must be enclosed in double quotation marks. To add a mnemonic to the text string, place the ampersand (&) ahead of the letter that will be the mnemonic. To use the ampersand as a character in a string, insert two ampersands (&&).

The *id* field takes a unique integer value that identifies the control.

The *x* and *y* fields take integer values that specify the *x* and *y* coordinates of the upper-left corner of the control. The coordinates are relative to the origin of the dialog box.

The *width* and *height* fields take integer values that specify the width and height of the control. The width units are 1/4 of the width of a character; the height units are 1/8 of the height of a character.

The optional *style* field contains WS_TABSTOP and/or WS_GROUP styles, which are fully described in Table 3.1, and/or BUTTON-class styles, which are fully described in Table 3.3. Styles can be combined using the bitwise OR operator.

Comments

The *x, y, width,* and *height* fields can use addition and subtraction operators (+ and −) for relative positioning. For example, "15 + 6" can be used for the *x* field.

Default style for **CHECKBOX** is BS_CHECKBOX, WS_TABSTOP.

Example

```
CHECKBOX "Arabic", 3, 10, 10, 40, 10
```

3.10.2.5 PUSHBUTTON Statement

Syntax

PUSHBUTTON ***text, id, x, y, width, height,*** ⟦*style*⟧

This statement defines a rectangle containing the given *text*. The control sends a message to its parent whenever the user clicks the mouse inside the rectangle.

The *text* field takes an ASCII string that specifies the text to be displayed. The string must be enclosed in double quotation marks. To add a mnemonic to the text string, place the ampersand (&) ahead of the letter that will be the mnemonic. To use the ampersand as a character in a string, insert two ampersands (&&).

The *id* field takes a unique integer value that identifies the control.

The *x* and *y* fields take integer values that specifies the *x* and *y* coordinates of the upper-left corner of the control. The coordinates are relative to the origin of the dialog box.

The *width* and *height* fields take integer values that specify the width and height of the control. The width units are 1/4 of the width of a character; the height units are 1/8 of the height of a character.

The optional *style* field contains WS_TABSTOP, WS_DISABLED, and/or WS_GROUP styles, which are fully described in Table 3.1, and/or BUTTON-class styles, which are fully described in Table 3.3. Styles can be combined using the bitwise OR operator.

Comments

The *x, y, width,* and *height* fields can use addition and subtraction operators (+ and −) for relative positioning. For example, "15 + 6" can be used for the *x* field.

Default style for **PUSHBUTTON** is BS_PUSHBUTTON, WS_TABSTOP.

Example

```
PUSHBUTTON "ON", 7, 10, 10, 20, 10
```

3.10.2.6 LISTBOX Statement

Syntax

LISTBOX ***id, x, y, width, height,*** [[*style*]]

This statement defines a list box belonging to the LISTBOX class. It creates a rectangle that contains a list of strings (such as filenames) from which the user can make selections.

The *id* field takes a unique integer value that identifies the control.

The *x* and *y* fields take integer values that specify the *x* and *y* coordinates of the upper-left corner of the control. The coordinates are relative to the origin of the dialog box.

The *width* and *height* fields take integer values that specify the width and height of the control. The width units are 1/4 of the width of a character; the height units are 1/8 of the height of a character.

The optional *style* field contains WS_BORDER and/or WS_VSCROLL styles, which are fully described in Table 3.1, and/or LISTBOX-class styles, which are fully described in Table 3.3. Styles can be combined using the bitwise OR operator.

Comments

The *x*, *y*, *width*, and *height* fields can use addition and subtraction operators (+ and –) for relative positioning. For example, "15 + 6" can be used for the *x* field.

Default style for **LISTBOX** is LBS_NOTIFY, LBS_SORT, WS_VSCROLL, WS_BORDER.

For information on the recommended keys for use in list-box controls, see the *Microsoft Windows Application Style Guide.*

Example

```
LISTBOX 666, 10, 10, 50, 54
```

3.10.2.7 GROUPBOX Statement

Syntax

GROUPBOX ***text*, *id*, *x*, *y*, *width*, *height*,** [[*style*]]

This statement defines a group box belonging to the BUTTON class. It creates a rectangle that groups other controls together. The controls are grouped by drawing a border around them and displaying the given text in the upper-left corner.

The *text* field takes an ASCII string that specifies the text to be displayed. The string must be enclosed in double quotation marks. To add a mnemonic to the text string, place the ampersand (&) ahead of the letter that will be the mnemonic. Selecting the mnemonic moves the input focus to the next control, in the order set in the resource file. To use the ampersand as a character in a string, insert two ampersands (&&).

The *id* field takes a unique integer value that identifies the control.

The *x* and *y* fields take integer values that specify the *x* and *y* coordinates of the upper-left corner of the control. The coordinates are relative to the origin of the dialog box.

The *width* and *height* fields take integer values that specify the width and height of the control. The width units are 1/4 of the width of a character; the height units are 1/8 of the height of a character.

The optional *style* field contains WS_TABSTOP or WS_DISABLED styles, which are fully described in Table 3.1, and/or BUTTON-class styles, which are fully described in Table 3.3. Styles can be combined using the bitwise OR operator.

Comments

The *x, y, width,* and *height* fields can use addition and subtraction operators (+ and −) for relative positioning. For example, "15 + 6" can be used for the *x* field.

Default style for **GROUPBOX** is BS_GROUPBOX, WS_TABSTOP.

Example

```
GROUPBOX "Output", 42, 10, 10, 30, 50
```

3.10.2.8 DEFPUSHBUTTON Statement

Syntax

DEFPUSHBUTTON *text, id, x, y, width, height,* [[*style*]]

This statement defines a default pushbutton control that belongs to the BUTTON class. It creates a small rectangle with a bold outline that represents the default response for the user. The text is displayed inside the button. The control highlights the button in the usual way when the user clicks the mouse in it and sends a message to its parent window.

The *text* field takes an ASCII string that specifies the text to be displayed. The string must be enclosed in double quotation marks. To add a mnemonic to the text string, place the ampersand (&) ahead of the letter that will be the mnemonic. To use the ampersand as a character in a string, insert two ampersands (&&).

The *id* field takes a unique integer value that identifies the control.

The *x* and *y* fields take integer values that specify the *x* and *y* coordinates of the upper-left corner of the control. The coordinates are relative to the origin of the dialog box.

The *width* and *height* fields take integer values that specify the width and height of the control. The width units are 1/4 of the width of a character; the height units are 1/8 of the height of a character.

The optional *style* field contains WS_TABSTOP, WS_GROUP and/or WS_DISABLED styles, which are fully described in Table 3.1, and/or BUTTON-class styles, which are fully described in Table 3.3. Styles can be combined using the bitwise OR operator.

Comments

The *x*, *y*, *width*, and *height* fields can use addition and subtraction operators (+ and −) for relative positioning. For example, "15 + 6" can be used for the *x* field.

Default style for **DEFPUSHBUTTON** is BS_DEFPUSHBUTTON, WS_TABSTOP.

Example

```
DEFPUSHBUTTON "ON", 7, 10, 10, 20, 10
```

3.10.2.9 RADIOBUTTON Statement

Syntax

RADIOBUTTON ***text, id, x, y, width, height,*** ⟦*style*⟧

This statement defines a radiobutton control belonging to the BUTTON class. It creates a small rectangle that has the given text displayed just to its right. The control highlights the button when the user clicks the mouse in it and sends a message to its parent window. The control removes the highlight and sends a message on the next click.

The *text* field takes an ASCII string that specifies the text to be displayed. The string must be enclosed in double quotation marks. To add a mnemonic to the text string, place the ampersand (&) ahead of the letter that will be the mnemonic. To use the ampersand as a character in a string, insert two ampersands (&&).

The *id* field takes a unique integer value that identifies the control.

The *x* and *y* fields take integer values that specify the *x* and *y* coordinates of the upper-left corner of the control. The coordinates are relative to the origin of the dialog box.

The *width* and *height* fields take integer values that specify the width and height of the control. The width units are 1/4 of the width of a character; the height units are 1/8 of the height of a character.

The optional *style* field contains WS_TABSTOP, WS_GROUP and/or WS_DISABLED styles, which are fully described in Table 3.1, and/or BUTTON-class styles, which are fully described in Table 3.3. Styles can be combined using the bitwise OR operator.

Comments

The *x, y, width,* and *height* fields can use addition and subtraction operators (+ and −) for relative positioning. For example, "15 + 6" can be used for the *x* field.

Default style for **RADIOBUTTON** is BS_RADIOBUTTON, WS_TABSTOP.

Example

```
RADIOBUTTON "AM 101", 10, 10, 10, 40, 10
```

3.10.2.10 EDITTEXT Statement

Syntax

EDITTEXT ***id, x, y, width, height,*** ⟦*style*⟧

This statement defines an EDIT control belonging to the EDIT class. It creates a rectangle in which the user can enter and edit text. The control displays a cursor when the user clicks the mouse in it. The user can then use the keyboard to enter text or edit the existing text. Editing keys include the BACKSPACE and DELETE keys. The user can also use the mouse to select the character or characters to be deleted, or to select the place to insert new characters.

The *id* field takes a unique integer value that identifies the control.

The *x* and *y* fields take integer values that specify the *x* and *y* coordinates of the upper-left corner of the control. The coordinates are relative to the origin of the dialog box.

The *width* and *height* fields take integer values that specify the width and height of the control. The width units are 1/4 of the width of a character; the height units are 1/8 of the height of a character.

The optional *style* field contains WS_TABSTOP, WS_GROUP, WS_VSCROLL, WS_HSCROLL, and/or WS_DISABLED styles, which are fully described in Table 3.1, and/or EDIT-class styles, which are fully described in Table 3.3. Styles can be combined using the bitwise OR operator. EDIT-class styles must not conflict.

Comments

The *x*, *y*, *width*, and *height* fields can use addition and subtraction operators (+ and –) for relative positioning. For example, "15 + 6" can be used for the *x* field.

Default style for **EDITTEXT** is WS_TABSTOP, ES_LEFT, WS_BORDER.

Keyboard use is predefined for edit controls. Predefined keys are listed in the *Microsoft Windows Application Style Guide.*

Example

```
EDITTEXT  3, 10, 10, 100, 10
```

3.10.2.11 ICON Statement

Syntax

ICON ***text*, *id*, *x*, *y*, *width*, *height*,** [[*style*]]

This statement defines an icon control belonging to the STATIC class. It creates an icon displayed in the dialog box. The given text is the name of an icon (not a filename) defined elsewhere in the resource file.

For the **ICON** statement, the *width* and *height* fields are ignored; the icon automatically sizes itself.

The *id* field takes a unique integer value that identifies the control.

The *x* and *y* fields take integer values that specify the *x* and *y* coordinates of the upper-left corner of the control. The coordinates are relative to the origin of the dialog box.

The *width* and *height* fields take integer values that specify the width and height of the control. The width units are 1/4 of the width of a character; the height units are 1/8 of the height of a character.

The optional *style* field allows only the SS_ICON style.

Comments

The *x*, *y*, *width*, and *height* fields can use addition and subtraction operators (+ and –) for relative positioning. For example, "15 + 6" can be used for the *x* field.

Default style for **ICON** is SS_ICON.

3.10.2.12 CONTROL Statement

Syntax

CONTROL ***text, id, class, style, x, y, width, height***

This statement defines a user-defined control window.

The *text* field takes an ASCII string that specifies the text to be displayed. The string must be enclosed in double quotation marks.

The *id* field takes a unique integer value that identifies the control.

The *class* field takes a predefined name, character string, or integer that defines the class. It can be any one of the control classes; for a list of the control classes, see Table 3.2. If it is a predefined name supplied by the application, it must be an ASCII string enclosed in double quotation marks.

The *style* field takes a predefined name or integer value that specifies the style of the given control. The exact meaning of *style* depends on the *class* value. Tables 3.2 and 3.3 list the control classes and corresponding styles.

The *x* and *y* fields take integer values that specify the *x* and *y* coordinates of the upper-left corner of the control. The coordinates are relative to the origin of the dialog box.

The *width* and *height* fields take integer values that specify the width and height of the control. The width units are 1/4 of the width of a character; the height units are 1/8 of the height of a character.

Comments

The *x*, *y*, *width*, and *height* fields can use addition and subtraction operators (+ and –) for relative positioning. For example, "15 + 6" can be used for the *x* field.

Table 3.2 describes the five control classes:

Table 3.2

Control Classes

Class	Description
BUTTON	A button control is a small rectangular child window that represents a "button" that the user can turn on or off by clicking it with the mouse. Button controls can be used alone or in groups, and can either be labeled or appear without text. Button controls typically change appearance when the user clicks them.
EDIT	An edit control is a rectangular child window in which the user can enter text from the keyboard. The user selects the control, and gives it the input focus, by clicking the mouse inside it or pressing the TAB key. The user can enter text when the control displays a flashing caret. The mouse can be used to move the cursor and select characters to be replaced, or to position the cursor for inserting characters. The BACKSPACE key can be used to delete characters. Edit controls use the fixed-pitch font and display ANSI characters. They expand tab characters into as many space characters as are required to move the cursor to the next tab stop. Tab stops are assumed to be at every eighth character position.
STATIC	Static controls are simple text fields, boxes, and rectangles that can be used to label, box, or separate other controls. Static controls take no input and provide no output.
LISTBOX	List-box controls consist of a list of character strings. The control is used whenever an application needs to present a list of names, such as filenames, that the user can view and select. The user can select a string by pointing to the string with the mouse and clicking a mouse button. When a string is selected, it is highlighted, and a notification message is passed to the parent window. A scroll bar can be used with a list-box control to scroll lists that are too long or too wide for the control window.

Table 3.2 *(continued)*

Class	Description
SCROLLBAR	A scroll-bar control is a rectangle that contains a scroll thumb and has direction arrows at both ends. The scroll bar sends a notification message to its parent whenever the user clicks the mouse in the control. The parent is responsible for updating the thumb position, if necessary. Scroll-bar controls have the same appearance and function as the scroll bars used in ordinary windows. Unlike scroll bars, scroll-bar controls can be positioned anywhere in a window and used whenever needed to provide scrolling input for a window. The scroll-bar class also includes Size-box controls. A Size-box control is a small rectangle that the user can expand to change the size of the window.

Table 3.3 describes the control styles for each of the control classes:

Table 3.3

Control Styles

Style	Description
BUTTON Class	
BS_PUSHBUTTON	A small elliptical button containing the given text. The control sends a message to its parent whenever the user clicks the mouse inside the rectangle.
BS_DEFPUSHBUTTON	A small elliptical button with a bold border. This button represents the default user response. Any text is displayed within the button. Windows sends a message to the parent window when the user clicks the mouse in this button.
BS_CHECKBOX	A small rectangular button that may be checked; its border becomes bold when the user clicks the mouse in it. Any text appears to the right of the button.
BS_AUTOCHECKBOX	Identical to BS_CHECKBOX except that the button automatically toggles its state whenever the user clicks it.

Table 3.3 *(continued)*

Style	Description
BS_RADIOBUTTON	A small circular button whose border becomes bold when the user clicks the mouse in it. In addition, to make the border bold, Windows sends a message to the button's parent notifying it that a click occurred. On the next click, Windows makes the border normal again and sends another message.
BS_AUTORADIOBUTTON	Identical to BS_RADIOBUTTON except that the button is checked, the application is notified with BN_CLICKED, and all other radio buttons in the group are unchecked.
BS_LEFTTEXT	Causes text to appear on the left side of the radio button or check-box button. Use this style with BS_CHECKBOX, BS_3STATE, or BS_RADIOBUTTON styles.
BS_3STATE	Identical to BS_CHECKBOX except that a button can be grayed as well as checked or unchecked. The grayed state is typically used to show that a check box has been disabled.
BS_AUTO3STATE	Identical to BS_3STATE except that the button automatically toggles its state when the user clicks it.
BS_GROUPBOX	A rectangle into which other buttons are grouped. Any text is displayed in the rectangle's upper-left corner.
BS_USERBUTTON	A user-defined button. The parent is notified when the button is clicked. Notification includes a request to paint, invert, and disable the button.
EDIT Class	
ES_LEFT	Flush-left text.
ES_CENTER	Centered text.
ES_RIGHT	Flush-right text.

Table 3.3 *(continued)*

Style	Description
ES_MULTILINE	Multiple-line edit control. (The default is single-line.) If the ES_AUTOVSCROLL style is specified, the edit control shows as many lines as possible and scrolls vertically when the user presses the ENTER key. (This is actually the carriage-return character, which the edit control expands to a carriage-return/line-feed combination. A line feed is not treated the same as a carriage return.) If ES_AUTOVSCROLL is not given, the edit control shows as many lines as possible and beeps if the user presses ENTER when no more lines can be displayed. If the ES_AUTOHSCROLL style is specified, the multiple-line edit control automatically scrolls horizontally when the caret goes past the right edge of the control. To start a new line, the user must press the ENTER key. If ES_AUTOHSCROLL is not given, the control automatically wraps words to the beginning of the next line when necessary; a new line is also started if the user presses the ENTER. The position of the wordwrap is determined by the window size. If the window size changes, the wordwrap position changes, and the text is redisplayed. Multiple-line edit controls can have scroll bars. An edit control with scroll bars processes its own scroll-bar messages. Edit controls without scroll bars scroll as described above, and process any scroll messages sent by the parent window.
ES_AUTOVSCROLL	Text is automatically scrolled up one page when the user presses the ENTER key on the last line.
ES_AUTOHSCROLL	Text is automatically scrolled to the right by 10 characters when the user types a character at the end of the line. When the user presses the ENTER key, the control scrolls all text back to position 0.
ES_NOHIDESEL	Normally, an edit control hides the selection when the control loses the input focus, and inverts the selection when the control receives the input focus. Specifying ES_NOHIDESEL overrides this default action.

Table 3.3 *(continued)*

Style	**Description**
STATIC Class	
SS_LEFT	A simple rectangle displaying the given text flush left in the rectangle. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next flush-left line.
SS_CENTER	A simple rectangle displaying the given text centered in the rectangle. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next centered line.
SS_RIGHT	A simple rectangle displaying the given text flush right in the rectangle. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next flush-right line.
SS_ICON	An icon displayed in the dialog box. The given text is the name of an icon (not a filename) defined elsewhere in the resource file. For the **ICON** statement, the *width* and *height* parameters in **CreateWindow** are ignored; the icon automatically sizes itself.
SS_BLACKRECT	Black-filled rectangle.
SS_GRAYRECT	Gray-filled rectangle.
SS_WHITERECT	White-filled rectangle.
SS_BLACKFRAME	Box with black frame.
SS_GRAYFRAME	Box with gray frame.
SS_WHITEFRAME	Box with white frame.
SS_USERITEM	User-defined item.
LISTBOX Class	
LBS_NOTIFY	The parent receives an input message whenever the user clicks or double-clicks a string.
LBS_MULTIPLESEL	The string selection is toggled each time the user clicks or double-clicks the string. Any number of strings can be selected.

Table 3.3 *(continued)*

Style	Description
LBS_SORT	The strings in the list box are sorted alphabetically.
LBS_NOREDRAW	The list-box display is not updated when changes are made. This style can be changed at any time by sending a WM_SETREDRAW message.
SCROLLBAR Class	
SBS_VERT	Vertical scroll bar. If neither SBS_RIGHTALIGN nor SBS_LEFTALIGN is specified, the scroll bar has the height, width, and position given in the **CreateWindow** function.
SBS_RIGHTALIGN	Used with SBS_VERT. The right edge of the scroll bar is aligned with the right edge of the rectangle specified by the *x, y, width,* and *height* values given in the **CreateWindow** function. The scroll bar has the default width for system scroll bars.
SBS_LEFTALIGN	Used with SBS_VERT. The left edge of the scroll bar is aligned with the left edge of the rectangle specified by the *x, y, width,* and *height* values given in the **CreateWindow** function. The scroll bar has the default width for system scroll bars.
SBS_HORZ	Horizontal scroll bar. If neither SBS_BOTTOMALIGN nor SBS_TOPALIGN is specified, the scroll bar has the height, width, and position given in the **CreateWindow** function.
SBS_TOPALIGN	Used with SBS_HORZ. The top edge of the scroll bar is aligned with the top edge of the rectangle specified by the *x, y, width,* and *height* values given in the **CreateWindow** function. The scroll bar has the default height for system scroll bars.
SBS_BOTTOMALIGN	Used with SBS_HORZ. The bottom edge of the scroll bar is aligned with the bottom edge of the rectangle specified by the *x, y, width,* and *height* values given in the **CreateWindow** function. The scroll bar has the default height for system scroll bars.

Table 3.3 *(continued)*

Style	Description
SBS_SIZEBOX	Size box. If neither SBS_SIZEBOX-BOTTOMRIGHTALIGN nor SBS_SIZEBOXTOPLEFTALIGN is specified, the Size box has the height, width, and position given in the **CreateWindow** function.
SBS_SIZEBOXTOPLEFTALIGN	Used with SBS_SIZEBOX. The top-left corner of the Size box is aligned with the top-left corner of the rectangle specified by the *x*, *y*, *width*, and *height* values given in the **CreateWindow** function. The Size box has the default size for system Size boxes.
SBS_SIZEBOXBOTTOMRIGHTALIGN	Used with SBS_SIZEBOX. The bottom-right corner of the Size box is aligned with the bottom-right corner of the rectangle specified by the *x*, *y*, *width*, and *height* values given in the **CreateWindow** function. The Size box has the default size for system Size boxes.

3.11 Directives

The resource directives are special statements that define actions to be performed 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.*

3.11.1 #include Statement

Syntax

# include *filename*

This directive copies the contents of the file specified by *filename* into your resource script before **rc** processes the script.

The *filename* field takes 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* field is handled as a C string, and 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 "windows.h"

PenSelect MENU
BEGIN
     Menuitem "&black pen", BLACK_PEN
END
```

3.11.2 #define Statement

Syntax

# define *name value*

This directive assigns the given *value* to *name*. All subsequent occurrences of *name* are replaced by *value*.

The *value* field takes any integer value, character string, or line of text.

Examples

```
#define     nonzero       1
#define     USERCLASS    "MyControlClass"
```

3.11.3 #undef Statement

Syntax

# undef *name*

This directive removes the current definition of *name*. All subsequent occurrences of *name* are processed without replacement.

Examples

```
#undef      nonzero
#undef      USERCLASS
```

3.11.4 #ifdef Statement

Syntax

#ifdef *name*

This directive carries out conditional compilation of the resource file by checking the specified *name*. If *name* has been defined using a **#define** directive, **#ifdef** directs the resource compiler to continue with the statement immediately after **#ifdef**. If *name* has not been defined, **#ifdef** directs the compiler to skip all statements up to the next **#endif** directive.

Example

```
#ifdef Debug
errbox BITMAP errbox.bmp
#endif
```

3.11.5 #ifndef Statement

Syntax

#ifndef *name*

This directive carries out conditional compilation of the resource file by checking the specified *name*. If *name* has not been defined or if its definition has been removed using the **#undef** directive, **#ifndef** directs the resource compiler to continue processing statements up to the next **#endif**, **#else**, or **#elif** directive, then skip to the statement after **#endif**. If *name* is defined, **#ifndef** directs the compiler to skip to the next **#endif**, **#else**, or **#elif** directive.

Example

```
#ifndef Optimize
errbox BITMAP errbox.bmp
#endif
```

3.11.6 #if Statement

Syntax

#if *constant-expression*

This directive carries out conditional compilation of the resource file by checking the specified *constant-expression.* If *constant-expression* is nonzero, **#if** directs the resource compiler to continue processing statements up to the next **#endif**, **#else**, or **#elif** directive, then skip to the statement after **#endif**. If *constant-expression* is zero, **#if** directs the compiler to skip to the next **#endif**, **#else**, or **#elif** directive.

The *constant-expression* field specifies a defined name, an integer constant, or an expression consisting of names, integers, and arithmetic and relational operators.

Example

```
#if Version<3
errbox BITMAP errbox.bmp
#endif
```

3.11.7 #elif Statement

#elif *constant-expression*

This directive marks an optional clause of a conditional compilation block defined by an **#ifdef**, **#ifndef**, or **#if** directive. The **#elif** directive carries out conditional compilation of the resource file by checking the specified *constant-expression.* If *constant-expression* is nonzero, **#elif** directs the resource compiler to continue processing statements up to the next **#endif**, **#else**, or **#elif** directive, then skip to the statement after **#endif**. If *constant-expression* is zero, **#elif** directs the compiler to skip to the next **#endif**, **#else**, or **#elif** directive. Any number of **#elif** directives can be used in a conditional block.

The *constant-expression* field specifies a defined name, an integer constant, or an expression consisting of names, integers, and arithmetic and relational operators.

Example

```
#if Version<3
errbox BITMAP errbox.bmp
#elif Version<7
errbox BITMAP userbox.bmp
#endif
```

3.11.8 #else Statement

Syntax

# else

This directive marks an optional clause of a conditional compilation block defined by an **#ifdef**, **#ifndef**, or **#if** directive. The **#else** directive must be the last directive before **#endif.**

Example

```
#ifdef Debug
errbox BITMAP errbox.bmp
#else
errbox BITMAP userbox.bmp
#endif
```

3.11.9 #endif Statement

Syntax

# endif

This directive marks the end of a conditional compilation block defined by an **#ifdef** directive. One **#endif** is required for each **#ifdef** directive.

Chapter 4
Windows Linker: Link4

4.1 Introduction

You create executable Microsoft 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 file that you can load and run with Windows.

This chapter describes how to use **link4**, how to create module-definition 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.*

4.2.1 Module Definitions for Applications

A module-definition file for an application must contain a **NAME** statement that defines 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 other 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 named "SampleWndProc" is the procedure exported so that Windows can call it. 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 that defines 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 that 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 Module-Definition Statements

The module-definition file contains one or more of the following module statements:

Statement	Description
NAME	Module name
LIBRARY	Library name
DESCRIPTION	One-line description of the module
DATA	Data-segment attributes
CODE	Code-segment attributes
HEAPSIZE	Local-heap size in bytes
STACKSIZE	Local-stack size in bytes
SEGMENTS	Additional code segment
EXPORTS	Exported functions
IMPORTS	Imported functions
STUB	Old-style executable

4.3.1 NAME Statement

Syntax

NAME *modulename*

This statement defines the name of the application's executable module. The name is used to identify the module when importing or exporting functions.

The *modulename* field specifies one or more ASCII characters.

Comments

The *modulename* field is optional. If the field is not given, the linker uses the filename part of the executable file (that is, the name with the extension removed).

If neither a **NAME** nor a **LIBRARY** statement is given in the definition file, the linker assumes that a **NAME** statement without a *modulename* field is desired.

Example

```
NAME Calendar
```

4.3.2 LIBRARY Statement

Syntax

LIBRARY *libraryname*

This statement defines the name of a library module. Library modules are resource modules that contain code, data, and other resources but are not intended to be executed as an independent program.

The *libraryname* field specifies one or more ASCII characters that define the name of the library module.

Comments

The start address of the module is determined by the object files. It is an internally defined function.

The *libraryname* field is optional. If the field is not given, the linker uses the filename part of the executable file (that is, the name with the extension removed).

Example

```
LIBRARY User
```

4.3.3 DESCRIPTION Statement

Syntax

DESCRIPTION '*text*'

This statement inserts text into the application's module. It is useful for embedding source-control or copyright information.

The *text* field specifies one or more ASCII characters. The string must be enclosed in single quotation marks.

Example

```
DESCRIPTION 'Microsoft Windows Template Application'
```

4.3.4 HEAPSIZE Statement

Syntax

HEAPSIZE *bytes*

This statement defines the number of bytes needed by the application for its local heap. An application uses the local heap whenever it allocates local memory.

The default heap size is zero.

The *bytes* field takes an integer value that specifies the heap size in bytes. It must not exceed 65,536 (the size of a single physical segment).

Example

```
HEAPSIZE 4096
```

4.3.5 STACKSIZE Statement

Syntax

STACKSIZE *bytes*

This statement defines the number of bytes needed by the application for its local stack. An application uses the local stack whenever it calls its own functions. A minimum stack size of 4096 bytes is recommended.

The default stack size is zero, if the application makes no function calls. Otherwise, it is 4096.

The *bytes* field takes an integer value that specifies the stack size in bytes.

Example

```
STACKSIZE 4096
```

4.3.6 CODE Statement

Syntax

CODE ⟦*segment-attributes*⟧

This statement defines the attributes of the standard code segment. The standard code segment is the application segment having the name _TEXT and belonging to the class CODE. In C applications, the standard segment is created automatically if no specific segment name is given in the C-compiler command line.

The *segment-attributes* field takes one or more optional keywords that specify the code-segment attributes. They can be any combination of the following:

Keyword	Description
FIXED	Segment remains at a fixed memory location.
MOVEABLE	Segment can be moved if necessary to compact memory.
DISCARDABLE	Segment can be discarded if no longer needed.
PRELOAD	Segment is loaded immediately.
LOADONCALL	Segment is loaded when called.
SHARED	Segment can be shared.
NONSHARED	Segment cannot be shared.
EXECUTEONLY	Segment can be executed only.
EXECUTEREAD	Segment can be read as data as well as executed.

Comments

If no **CODE** statement is given in the module-definition file, the default attributes for the segment are **MOVEABLE**, **PRELOAD**, **NONSHARED**, and **EXECUTEREAD**.

If a **CODE** statement is given, the default attributes are **FIXED**, **LOADONCALL**, **NONSHARED**, and **EXECUTEREAD**, unless these are explicitly overridden.

If conflicting options are given in the same statement, **link4** uses the overriding option to determine the segment attributes. **MOVEABLE** overrides **FIXED**; **PRELOAD** overrides **LOADONCALL**; **SHARED** overrides **NONSHARED**; and **EXECUTEONLY** overrides **EXECUTEREAD**.

SHARED, **NONSHARED**, **EXECUTEONLY**, and **EXECUTEREAD** are used for 80286 protected-mode programs only.

PURE and **IMPURE** are alternate keywords that can be used in place of **SHARED** and **NONSHARED**, respectively.

Example

```
CODE MOVEABLE LOADONCALL
```

4.3.7 DATA Statement

Syntax

DATA [[*segment-attributes*]]

This statement defines the attributes of the standard data segment. The standard data segment is all application segments belonging to the group DGROUP and the class DATA. In C applications, the standard data segment is created automatically.

The *segment-attributes* field takes one or more optional keywords that specify the attributes of the data segment. They can be any combination of the following:

Keyword	Description
NONE	There is no data segment.
SINGLE	A single segment is shared by all instances of the module (valid only for library modules).

MULTIPLE	One segment exists for each instance.
FIXED	Segment remains at a fixed memory location.
MOVEABLE	Segment can be moved if necessary to compact memory.
DISCARDABLE	Segment can be discarded if no longer needed.
PRELOAD	Segment is loaded immediately.
LOADONCALL	Segment is loaded when accessed.
SHARED	Segment contains data that does not change during execution.
NONSHARED	Segment contains data that may change during execution.
READONLY	Segment contents can be read only.
READWRITE	Segment contents can be read and modified.

Comments

If no **DATA** segment is given in the module-definition file, the default attributes for the segment are **MULTIPLE**, **MOVEABLE**, **PRELOAD**, **NONSHARED**, and **READWRITE**.

If a **DATA** statement is given, the default attributes are **MULTIPLE**, **FIXED**, **LOADONCALL**, **NONSHARED**, and **READWRITE**, unless these are explicitly overridden.

If conflicting options are given in the same statement, **link4** uses the overriding option to determine the segment attributes. **MULTIPLE** overrides **NONE** and **SINGLE**; **SINGLE** overrides **NONE**; **MOVEABLE** overrides **FIXED**; **PRELOAD** overrides **LOADONCALL**; **SHARED** overrides **NONSHARED**; and **READONLY** overrides **READWRITE**.

The **SINGLE** option implies **SHARED**, and vice versa; **MULTIPLE** implies **NONSHARED**, and vice versa. **Link4** ignores **SHARED** if it is used with **MULTIPLE** and ignores **NONSHARED** if it is used with **SINGLE**.

SHARED, **NONSHARED**, **READONLY**, and **READWRITE** are used for 80286 protected-mode programs only.

PURE and **IMPURE** are alternate keywords that can be used in place of **SHARED** and **NONSHARED**, respectively.

Example

```
DATA MOVEABLE SINGLE
```

4.3.8 SEGMENTS Statement

Syntax

SEGMENTS *segmentname* ⟦**CLASS** '*class-name*'⟧ ⟦*minalloc*⟧ ⟦*segment-attributes*⟧

This statement defines the segment attributes of additional code and data segments.

The *segmentname* field specifies a character string that names the new segment. It can be any name, including the standard segment names _TEXT and _DATA that represent the standard code and data segments.

The *class-name* field takes an optional keyword that specifies the class name of the given segment. If no class name is given, **link4** assumes the class name CODE by default.

The *minalloc* field takes an integer value that specifies the minimum allocation size for the segment.

The *segment-attributes* field takes one or more optional keywords that specify the attributes of the given segment. They can be any combination of the following:

Keyword	Description
FIXED	Segment remains at a fixed memory location.
MOVEABLE	Segment can be moved if necessary to compact memory.
DISCARDABLE	Segment can be discarded if no longer needed.
SHARED	Segment can be shared.
NONSHARED	Segment cannot be shared.
PRELOAD	Segment is loaded immediately.
LOADONCALL	Segment is loaded when accessed or called.
EXECUTEONLY	Segment can be executed only.
EXECUTEREAD	Segment can be read as data as well as executed.
READONLY	Segment contents can be read only.
READWRITE	Segment contents can be read and modified.

Comments

If no **SEGMENTS** statement is given in the module-definition file, the default attributes for nonstandard segments are **MOVEABLE**, **PRELOAD**, **NONSHARED**, and **EXECUTEREAD**.

If a **SEGMENTS** statement is given but only a segment name and class are given, the default attributes are **FIXED**, **LOADONCALL**, **NONSHARED**, and **EXECUTEREAD** or **READWRITE**, unless these are explicitly overridden.

If conflicting options are given in the same statement, **link4** uses the overriding option to determine the segment attributes. **MOVEABLE** overrides **FIXED**; **PRELOAD** overrides **LOADONCALL**; **SHARED** overrides **NONSHARED**; **EXECUTEONLY** overrides **EXECUTEREAD**; and **READONLY** overrides **READWRITE**.

SHARED, **NONSHARED**, **EXECUTEONLY**, and **EXECUTEREAD** are used for 80286 protected-mode programs only.

PURE and **IMPURE** are alternate keywords that can be used in place of **SHARED** and **NONSHARED**, respectively.

Example

```
SEGMENTS
      _TEXT FIXED
      _INIT PRELOAD DISCARDABLE
      _RES  CLASS 'DATA' PRELOAD DISCARDABLE
```

4.3.9 EXPORTS Statement

Syntax

EXPORTS *exportname* ⟦*ordinal-option*⟧ ⟦*res-option*⟧ ⟦*data-option*⟧ ⟦*parameter-option*⟧

This statement defines the names and attributes of the functions to be exported to other applications. The **EXPORTS** keyword marks the beginning of the definitions. It can be followed by any number of export definitions, each on a separate line.

The *exportname* field specifies one or more ASCII characters that defines the function name. It has the following form:

<*entryname*> ⟦=*internalname*⟧

where the *entryname* field specifies the name to be used by other applications to access the exported function, and *internalname* is an optional field that defines the actual name of the function if *entryname* is not the actual name.

The optional *ordinal-option* field defines the function's ordinal value. It has the following form:

@ *ordinal*

where *ordinal* takes an integer value that specifies the function's ordinal value. The ordinal value defines the location of the function's name in the application's string table.

The *res-option* field takes the optional keyword **RESIDENTNAME**, which specifies that the function's name must be resident at all times.

The *data-option* field takes the optional keyword **NODATA**, which specifies that the function is not bound to a specific data segment. When invoked, the function uses the current data segment.

The *parameter-option* field takes an optional integer value that specifies the number of words the function expects to be passed as parameters.

Example

```
EXPORTS
        SampleRead=read2bin @1 8
        StringIn=str1 @2 4
        CharTest NODATA
```

4.3.10 IMPORTS Statement

Syntax

IMPORTS [[*internal-option*]] *modulename* [[*entry-option*]]

This statement defines the names and attributes of the functions to be imported from other applications. The **IMPORTS** keyword marks the beginning of the definitions. It can be followed by any number of import definitions, each on a separate line.

The optional *internal-option* field specifies the name that the application will use to call the function. It has the following form:

internal-name=

where *internal-name* is one or more ASCII characters. This name must be unique.

The *modulename* field specifies the name of the executable module that contains the function.

The optional *entry-option* field specifies the function to be imported. It can be one of the following:

.entryname
.entryordinal

where *entryname* is the actual name of the function, and *entryordinal* is the ordinal value of the function.

Example

```
IMPORTS
          Sample.SampleRead
          write2hex=Sample.SampleWrite
          Read.1
```

4.3.11 STUB Statement

Syntax

STUB *'filename'*

This statement appends the old-style executable file given by *filename* to the beginning of the module. The executable stub should display a warning message and terminate if the user attempts to execute the module without having loaded Windows. The default file *winstub.exe* can be used if no other actions are required.

The *filename* field specifies the name of the old-style executable file that will be appended to the module. The name must have the DOS filename format.

Comments

If the file named by *filename* is not in the current directory, the linker searches for the file in the directories specified by the user's PATH environment variable.

Example

```
STUB 'winstub.exe'
```

4.4 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** program 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.

4.4.1 Link4 Command

Syntax

link4 ⟦*options*⟧ *object-files*,⟦*exe-file*⟧,⟦*map-file*⟧,⟦*lib-files*⟧,*def-file*

The *options* parameter specifies one or more keywords (described in Section 4.4.2) that direct **link4** to carry out special operations.

The *object-files* parameter 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 that you can give more than one *object-file* if necessary. Multiple filenames must be separated by spaces or the plus sign (+).

The *exe-file* parameter specifies the name you want the executable file to have.

The *map-file* parameter specifies the name you want the map file to have.

The *lib-files* parameter specifies the names of Windows or standard-language libraries.

The *def-file* parameter 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** program uses default filename extensions if you do not explicitly provide extensions. Thus, in the preceding example, the filename *sample* is extended to *sample.obj.* Library names are extended with the *.lib* extension.

4.4.2 Link4 Options

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

Option	Description
/alignment:*size*	This option directs **link4** to align segment data in the executable file along the boundaries specified by *size.* The *size* parameter 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* parameter 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 is typically 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 is typically 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 size limit of the segment 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 is typically 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.

4.5 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 be specified on the **link4** command line of the applications that use that library.

Syntax

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

The *imp-lib-name* parameter specifies the name you want the new import library to have.

The *mod-def-file* parameter specifies the name of the module-definition file for the Windows library.

Example

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

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

4.6 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.

Syntax

exehdr *exe-filename*

The *exe-filename* parameter specifies the name of any file with a *.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 variable, be sure the development directory is listed first.) Then **symdeb** will automatically 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	Specifies 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	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	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

```
cl -d -c -AS -Gsw -Os -Zdp test.c
link4 test,test,test/map/li,slibw, test
mapsym test
```

5.2.3 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 are automatically 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 can either 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* parameter specifies one or more **symdeb** options. The *symbolfiles* parameter specifies the names of symbol files. The *arguments* parameter specifies arguments 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.

Section 5.4.1 describes 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* parameter 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* parameter specifies the macro, and *command-string* specifies 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 PC, because **symdeb** automatically checks the hardware on startup. If **symdeb** does not find that the hardware is an IBM PC, 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 hyphen (–), and the option letter can be given with either uppercase or lowercase letters.

Files containing a hyphen in the filename must be renamed before use with **symdeb**. Otherwise, **symdeb** will interpret the hyphen 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's symbol file before any other symbol files. This ensures that the application's symbol file is open and that 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 symbol files for the kernel, user, and GDI libraries, *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 is typically used to temporarily stop 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 PC AT only.)

CONTROL+C	Cancels the current **symdeb** command. This key combination does not 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 must first 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 is currently 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 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 can also 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 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* field specifies the name of the application or library to receive the allocated memory. The *segment-name* field specifies the name of the code or data segment within the application or library that will occupy the memory block. The *segment-address* field specifies 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* fields specify 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* field specifies 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 is actually 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

If a symbol file does not contain line-number information, the **v**, **s+**, and **s&** 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 (Control) 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 5.1 is a summary of **symdeb** command syntax and purpose:

Table 5.1

Symdeb Commands

Syntax	Meaning
a ⟦*address*⟧	Assemble
ba ⟦*mode*⟧⟦*size*⟧**address**⟦*value*⟧⟦*command-string*⟧	Sets address breakpoint(s) on 80386 machines.
bc ⟦*id-list*⟧	Clear breakpoint(s)

Table 5.1 *(continued)*

Syntax	Meaning
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
kt *pdb* ⟦*value*⟧	Backtrace task
l ⟦*address* ⟦*drive record count*⟧⟧	Load

Table 5.1 *(continued)*

Syntax	Meaning
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, they 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, they 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 a 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 an 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 symbol (@), 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 65,535. 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 Address Command

Syntax

ba *option size address* ⟦*value*⟧ ⟦*command-string*⟧

This command, used with 80386 machines, sets an address breakpoint at a given address. If your program accesses memory at this address, **symdeb** will stop execution and display the current values of all registers and flags. There are three types of breakpoints you can set with the *option* parameter. Since the breakpoint address is a physical address, breakpoints should not be set for memory that might be moved (though you can temporarily lock the segment for the purpose of debugging). If **I** is specified, **symdeb** takes a breakpoint when the CPU fetches an instruction from the given *address*. If **R** is specified, **symdeb** takes a breakpoint when the CPU reads a byte, word, or double-word from the given *address*. If **U** is specified, **symdeb** takes a breakpoint when the CPU reads or writes a byte, word, or double-word to the given *address*.

The *size* parameter specifies the size of the data that **symdeb** expects to find read or written at the given *address*; the breakpoint will be taken only if the data has that size. If **B** is specified (8-bit byte), the command will break only at one address (for example, 0:10). If **W** is specified (16-bit word), the command will break at one of two addresses within that range (for example, 0:10 or 0:11 will cause a break within that word). If **D** is specified (32-bit double-word), the command will break at one of four addresses within that range (for example, 0:08, 0:09, 0:10, or 0:11 will cause a break within that double-word).

The *address* parameter can specify any valid address. The address value is rounded down if necessary to the nearest byte, word, or double-word boundary (for example, if a double-word address of 0:14 was requested, the command would access the address of the nearest double-word boundary below the address, in this case 0:12).

The optional *value* parameter specifies the number of times the breakpoint is to be ignored before being taken. It can be any 16-bit value.

The *command-string* parameter specifies 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.

The **bc**, **bd**, **be**, and **bl** commands can all be used on these breakpoints.

5.9.6 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 the wildcard character (*) is given, the command removes all breakpoints.

5.9.7 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 the wildcard character (*) is given, the command disables all breakpoints.

5.9.8 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 the wildcard character (*) is given, the command enables all breakpoints.

5.9.9 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.

5.9.10 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 optional *id* parameter specifies which breakpoint is to be created. If no *id* is given, the first available breakpoint number is used. The *address* parameter can be any valid instruction address (that is, it must be the first byte of an instruction). The optional *value* parameter specifies the number of times the breakpoint is to be ignored before being taken. It can be any 16-bit value. The optional *command-string* parameter specifies a 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 (;).

5.9.11 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.

5.9.12 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**.

5.9.13 Dump ASCII Command

Syntax

da ⟦*range*⟧

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

5.9.14 Dump Bytes Command

Syntax

db ⟦*range*⟧

This command displays the hexadecimal and ASCII values of the bytes in the given *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 are immediately followed by the corresponding ASCII values. The eighth and ninth hexadecimal values are separated by a hyphen (–). Nonprintable ASCII values are displayed as periods (.).

5.9.15 Dump Double-Words Command

Syntax

dd ⟦*range*⟧

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

5.9.16 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* field specifies the segment address of the first byte of the memory object. The *size* field specifies the size in paragraphs (multiples of 16 bytes) of the object. The *segment-type* field specifies the type of object. The type 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* field 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.

5.9.17 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* field specifies the address offset from the beginning of the data segment to the local memory object. The *size* field 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.

5.9.18 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 given *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.

5.9.19 Dump Task Queue Command

Syntax

dq

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

5.9.20 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 given *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 Ten-Byte Reals Command

Syntax

dt ⟦*range*⟧

This command displays the hexadecimal and decimal values of the ten-byte floating-point numbers in the given *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.

5.9.22 Dump Words Command

Syntax

dw ⟦*range*⟧

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

5.9.23 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**.) The default is **eb** (bytes). 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.

5.9.24 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 *address* and prompts for a replacement. If *list* is given, the command replaces the bytes at *address*, then displays the next byte and prompts for a replacement.

5.9.25 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 from the command, press the ENTER key.

5.9.26 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 *address* and prompts for a replacement. If *value* is given, the command replaces the double-word at *address*, then displays the next double-word and prompts for a replacement. Double-words must be typed as two words separated by a colon.

5.9.27 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 *address* and prompts for a replacement. If *value* is given, the command replaces the long-real value at *address,* then displays the next long-real value and prompts for a replacement.

5.9.28 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 *address* and prompts for a replacement. If *value* is given, the command replaces the short-real value at *address,* then displays the next short-real value and prompts for a replacement.

5.9.29 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 *address* and prompts for a replacement. If *value* is given, the command replaces the ten-byte real value at *address,* then displays the next ten-byte real value and prompts for a replacement.

5.9.30 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 *address* and prompts for a replacement. If *value* is given, the command replaces the word at *address*, then displays the next word and prompts for a replacement.

5.9.31 Fill Command

Syntax

f *range list*

This command fills the addresses in the given *range* with the values in *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.

5.9.32 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 *break address* 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 *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.

5.9.33 Hex Command

Syntax

h *value1 value2*

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

5.9.34 Input Command

Syntax

i *value*

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

5.9.35 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.

5.9.36 Backtrace Task Stack Command

Syntax

kt *pdb* ⟦*value*⟧

This command displays the stack frame of the program specified by *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* parameter must specify the segment address of the program descriptor block for the task to be traced.

5.9.37 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 *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:**5C). 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 2 (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, the **l** command 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.

5.9.38 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.

5.9.39 Macro Command

Syntax

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

This command defines or executes a **symdeb** command macro. The *id* parameter 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.

5.9.40 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* is 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.

5.9.41 Output Command

Syntax

o *value byte*

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

5.9.42 Program Step Command

Syntax

p ⟦=*startaddress*⟧ ⟦*value*⟧

This command executes the instruction at *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.

5.9.43 Quit Command

Syntax

q

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

5.9.44 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, all 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.

5.9.45 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.

5.9.46 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*.

5.9.47 Trace Command

Syntax

t ⟦=*startaddress*⟧ ⟦*value*⟧

This command executes the instruction at *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.

5.9.48 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.

5.9.49 View Command

Syntax

v *range*

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

5.9.50 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 *drive* to any number in the range zero (drive A) to 2 (drive C), set *record* to the first logical record to receive the data (a 1- to 4-digit hexadecimal number), and set *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.

5.9.51 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 no parameter is 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 specified, 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* parameter must specify 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* parameter must specify 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* parameter must specify 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.

5.9.52 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* parameter must specify 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.

5.9.53 Set Symbol Value Command

Syntax

z *symbol value*

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

5.9.54 Display Help Command

Syntax

?

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

5.9.55 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* parameter can specify any combination of numbers, symbols, addresses, and operators.

5.9.56 Source-Line Display Command

Syntax

.

This command displays the current source line.

5.9.57 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.

5.9.58 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.

5.9.59 Redirect Input and Output Commands

Syntax

=*filename*
~ *filename*

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

5.9.60 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 *dos-command* is given, **symdeb** passes the command to *command.com* for execution, then receives control back as soon as the command is completed.

5.9.61 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/dependent descriptions. Each description has the following general form:

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

The *targetfile* parameter specifies the name of a file that may need updating. The *dependentfiles* parameter specifies the name of a file on which the target file depends.

The names specified by the *targetfile* and *dependentfiles* parameters must be valid filenames. A pathname must be provided for any file that is not on the same drive and in the same 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* parameter can specify 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 number sign (#) 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 dependent.

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

Syntax

make ⟦*options*⟧ ⟦*macrodefinitions*⟧ *filename*

The *options* parameter specifies one or more of the options described in Section 6.2.3. The *macrodefinitions* parameter specifies one or more of the macro definitions described in Section 6.2.4. The *filename* parameter specifies 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	Description
/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* parameter is converted to uppercase: "flags" and "FLAGS" are equivalent. If you define a macro name but leave the *value* parameter blank, *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* parameter is also 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 can 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 could then start **make** with the following command line:

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

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

```
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

```
.asm.obj:
       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 you can make a study of their interaction. 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
       link4 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 **link4**; *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 **link4** command because *test.exe* is now out-of-date with respect to *math.lib.* It will also execute **mapsym** because *test.map* is created by **link4**.

Chapter 7
Assembly-Language Macros

7.1 Introduction

This chapter describes the **Cmacro** macros, a set of assembly-language macros that can be used with the Microsoft Macro Assembler (MASM) to create assembly-language Windows applications. The **Cmacros** provide a simplified interface to the function and segment conventions of high-level languages, such as C and Pascal.

The **Cmacros** are divided into the following groups:

Segment macros
Storage-allocation macros
Function macros
Call macros
Special-definition macros
Error macros

The following sections describe each group in detail.

7.2 CMACROS.INC File

The file *cmacros.inc* contains the assembly-language definitions for all the **Cmacro** macros. You must include this file at the beginning of the assembly-language source file by using the **INCLUDE** directive. The line has the following form:

```
INCLUDE cmacros.inc
```

You must give the full pathname if the macro file is not in the current directory or in a directory specified on the command line.

7.3 Cmacros Options

The **Cmacros** provide assembly-time options that define the memory model and the calling conventions that the application will use. The options must be selected in the assembly-language source file prior to the **INCLUDE** directive.

7.3.1 Memory-Model Selection

The memory-model options specify the memory model that the application will use. The memory model defines how many code and data segments are in the application. The following is a list of the possible memory models:

Model	Description
Small	One code segment and one data segment
Medium	Multiple code segments and one data segment
Compact	One code segment and multiple data segments
Large	Multiple code and data segments
Huge	Multiple code segments and multiple data segments with one or more data items larger than 64K

You select a memory model by defining the option name at the beginning of the assembly-language source file. The following Table 7.1 shows the option names available:

Table 7.1

Memory Options

Option Name	Memory Model	Code Size	Data Size
memS	small	small	small
memM	medium	large	small
memC	compact	small	large
memL	large	large	large
memH	huge	large	large

You can define a name by using the **EQU** directive. The definition has the following form:

```
memM      EQU      1
```

If no option is selected, the default is model is small.

When you select a memory-model option, two symbols are defined. These two symbols can be used for code that is dependent on the memory model:

SizeC	0 = small code	1 = large code	
SizeD	0 = small data	1 = large data	2 = huge data

7.3.2 Calling Conventions

The calling-convention option specifies the high-level-language calling convention that the application will use. You can select the calling convention by defining the value of the symbol **?PLM**. The following Table 7.2 lists the values and conventions:

Table 7.2
Calling Conventions

?PLM value	Convention	Description
0	Standard C	The caller pushes the rightmost argument onto the stack first, the leftmost last. The caller pops the arguments off the stack after control is returned.
1	Pascal	The caller pushes the leftmost argument onto the stack first, the rightmost last. The called function pops the arguments off the stack.

You can set the **?PLM** symbol value by using the **=** directive. The statement has the following form:

```
?PLM = 1
```

The default is the Pascal convention.

7.3.3 Windows Prolog/Epilog

The Windows prolog/epilog option specifies whether or not special prolog and epilog code should be used with each function. This special code defines the current data segment for the given function and is required for Windows applications.

You select this option by defining the value of the symbol **?WIN**. The following Table 7.3 lists the values:

Table 7.3

Prolog/Epilog Code Options

?WIN value	Meaning
0	Disables the special prolog/epilog code.
1	Enables the special prolog/epilog code.

You can set the **?WIN** symbol value by using the = directive. The statement has the following form:

```
?WIN = 1
```

The default is to have the prolog/epilog enabled.

7.3.4 Stack-Checking Option

You can enable stack checking by defining the symbol **?CHKSTK**. When stack checking is enabled, the prolog code calls the externally defined routine **CHKSTK** to allocate local variables.

You can define the **?CHKSTK** symbol by using the = directive. The statement has the following form:

```
?CHKSTK = 1
```

Once **CHKSTK** is defined, stack checking is enabled for the entire file.

The default (when **CHKSTK** is not defined) is no stack checking.

7.4 Segment Macros

The segment macros give access to the code and data segments that an application will use. These segments have the names, attributes, classes, and groups required by Windows.

The **Cmacros** have two predefined segments, named **CODE** and **DATA**, that any application can use without special definition. Medium-, large-, and huge-model applications can define additional segments by using the **createSeg** macro.

Syntax

createSeg *segName, logName, align, combine, class*

This macro creates a new segment that has the specified name and segment attributes. The macro automatically creates an **assumes** macro and an **OFFSET** macro for the new segment. This macro is intended to be used in medium-model Windows applications to define non-resident segments. The *segName* parameter specifies the actual name of the segment. This name is passed to the linker.

The *logName* parameter specifies the logical name of the segment. This name is used in all subsequent **sBegin**, **sEnd**, and **assumes** macros that refer to the segment.

The *align* parameter specifies the alignment type. It can be any one of the following:

BYTE
WORD
PARA
PAGE

The *combine* parameter specifies the combine type for the segment. It can be any one of the following:

PUBLIC
STACK
MEMORY
COMMON

If no combine type is given, a private segment is assumed.

The *class* parameter specifies the class name of the segment. The class name defines which segments must be loaded in consecutive memory.

Example

```
createSeg   _INIT,INITCODE,BYTE,PUBLIC,CODE

sBegin  INITCODE
assumes CS:INITCODE

       mov ax,initcodeOFFSET  sample

sEnd    INITCODE
```

Comments

The alignment, combine type, and class name are described in detail in the *Microsoft Macro Assembler Reference Manual.*

Syntax

sBegin *segName*

This macro opens up a segment. It is similar to the **SEGMENT** assembler directive.

The *segName* parameter specifies the name of the segment to be opened. It can be one of the predefined segments, **CODE** or **DATA**, or the name of a user-defined segment.

Examples

```
sBegin DATA
sBegin CODE
```

Syntax

sEnd ⟦*segName*⟧

This macro closes a segment. It is similar to the **ENDS** assembler directive.

The optional *segName* parameter specifies a name used for readability. If it is given, it must be the same as the name given in the matching **sBegin** macro.

Examples

```
sEnd
sEnd DATA
```

Syntax

assumes *segReg, segName*

This macro makes all references to data and code in the segment *segName* relative to the segment register given by *segReg.* It is similar to the **ASSUME** assembler directive.

The *segReg* parameter specifies the name of a segment register.

The *segName* parameter specifies the name of a predefined segment, **CODE** or **DATA**, or a user-defined segment.

Examples

```
assumes CS, CODE
assumes DS, CODE
```

Syntax

dataOFFSET *arg*

This macro generates an offset relative to the start of the group to which the **DATA** segment belongs. It is similar to the **OFFSET** assembler operator, but automatically provides the group name. For this reason, it should be used instead of **OFFSET**.

The *arg* parameter specifies a label name or offset value.

Example

```
mv ax,dataOFFSET label
```

Syntax

codeOFFSET*arg*

This macro generates an offset relative to the start of the group to which the **CODE** segment belongs. It is similar to the **OFFSET** assembler operator, but automatically provides the group name. For this reason, it should be used instead of **OFFSET**.

The *arg* parameter specifies a label name or offset value.

Example

```
mv ax,codeOFFSET label
```

Syntax

*segName***OFFSET** *arg*

This macro generates an offset relative to the start of the group to which the user-defined segment *segName* belongs. It is similar to the **OFFSET** assembler operator, but automatically provides the group name. For this reason, it should be used instead of **OFFSET**.

The *arg* parameter specifies a label name or offset value.

Example

```
mv ax,initcodeOFFSET label
```

7.4.1 Storage-Allocation Macros

These macros allocate static memory (either private or public), declare externally defined memory and procedures, and allow the definition of public labels.

Syntax

static*X* *name*, [[*initialValue*]], [[*replication*]]

This macro allocates private static-memory storage.

The *X* parameter specifies the size of storage to be allocated. It can be any one of the following:

Type	Description
B	Byte
W	Word
D	Double-word
Q	Quad-word
T	Ten bytes
CP	Code pointer (one word for small and compact models)
DP	Data pointer (one word for small and medium models)

The *name* parameter specifies the reference name of the allocated memory.

The optional *initialValue* parameter specifies an initial value for the storage. If no value is specified, the default is zero.

The optional *replication* parameter specifies a count of the number of times the allocation is to be duplicated. This parameter generates the **DUP** assembler operator.

Examples

```
staticW  flag,1
staticB  string, , 30
```

Syntax

global*X* ***name,*** ⟦***initialValue***⟧**,** ⟦***replication***⟧

This macro allocates public static-memory storage.

The *X* parameter specifies the size of the storage to be allocated. It can be any one of the following:

Type	Description
B	Byte
W	Word
D	Double-word
Q	Quad-word
T	Ten bytes
CP	Code pointer (one word for small and compact models)
DP	Data pointer (one word for small and medium models)

The *name* parameter specifies the reference name of the allocated memory.

The optional *initialValue* parameter specifies an initial value for the storage. If no value is specified, the default is zero.

The optional *replication* parameter specifies a count of the number of times the allocation is to be duplicated. This parameter generates the **DUP** assembler operator.

Examples

```
globalW  flag,1
globalB  string,0, 30
```

Syntax

extern*X* <*namelist*>

This macro defines one or more names that will be the labels of external variables or functions.

The *X* parameter specifies the storage size or function type. It can be any one of the following:

Type	Description
B	Byte
W	Word
D	Double-word
Q	Quad-word
T	Ten bytes
CP	Code pointer (one word for small and compact models)
DP	Data pointer (one word for small and medium models)
NP	Near function pointer
FP	Far function pointer
P	Near for small and compact models; far for other models

The *namelist* parameter specifies the list of the names of the variables or functions.

Examples

```
externB <DataBase>
externFP <SampleRead>
```

Syntax

label*X* <*namelist*>

This macro defines one or more names that will be the labels of public (global) variables or functions.

The *X* parameter specifies the storage size or function type. It can be any one of the following:

Type	Description
B	Byte
W	Word
D	Double-word
Q	Quad-word
T	Ten bytes
CP	Code pointer (one word for small and compact models)
DP	Data pointer (one word for small and medium models)
NP	Near function pointer
FP	Far function pointer
P	Near for small and compact models; far for other models

The *namelist* parameter specifies the list of the names of the external variables or functions.

Examples

```
labelB <DataBase>
labelFP <SampleRead>
```

7.4.2 Function Macros

The function macros define the names, attributes, parameters, and local variables of functions.

Syntax

cProc *procName*, <*attributes*>, <*autoSave*>

This macro defines the name and attributes of a function.

The *procName* parameter specifies the name of the function.

The *attributes* parameter specifies the function type. It can be a combination of the following:

Type	Description
NEAR	A near function. It can only be called from the segment in which it is defined.
FAR	A far function. It can be called from any segment.
PUBLIC	A public function. It can be externally declared in other source files.

The default attribute is **NEAR** and private (i.e., cannot be declared externally in other source files). The **NEAR** and **FAR** attributes cannot be used together. If more than one attribute is selected, the angle brackets are required.

The *autoSave* parameter specifies a list of registers to be saved when the function is invoked, and restored when exited. Any of the 8086's registers can be specified.

Comments

The C calling conventions require that the **si** and **di** registers be saved before altering their contents.

The **bp** register is always saved, regardless of whether it is present in the *autoSave* list.

Examples

```
cProc proc1, <FAR, ds,es>
cProc proc2, <NEAR,PUBLIC>
cProc proc3,,ds
```

Syntax

parm*X* <*namelist*>

This macro defines one or more function parameters. The parameters provide access to the arguments passed to the function. Parameters must appear in the same order as the arguments in the function call.

The *X* parameter specifies the storage size. It can be any one of the following:

Type	Description
B	Byte (allocated on a word boundary on the stack)
W	Word (allocated on a word boundary)
D	Double-word (allocated on a word boundary)
Q	Quad-word (aligned on a word boundary)
T	Ten-byte word (aligned on a word boundary)
CP	Code pointer (one word for small and compact models)
DP	Data pointer (one word for small and medium models)

The *namelist* parameter specifies the list of the parameter names.

Comments

The **parmD** macro creates two additional symbols, **OFF_** *name* and **SEG_** *name.* **OFF_** *name* is the offset portion of the parameter; **SEG_** *name* is the segment portion.

Only the parameter name is required when referring to the corresponding argument. Write your code like this:

```
mov     al,var1
```

Not like this:

```
mov     al,byte ptr var1[bp]
```

Examples

```
parmW var1
parmB <var2,var3,var4>
parmD <var5>
```

Syntax

local*X* **<***namelist***>,** *size*

This macro defines one or more frame variables for the function. To keep the words in the stack aligned, the macro ensures that the total space allocated is an even number of bytes.

The *X* parameter specifies the storage size. It can be any one of the following:

Type	Description
B	Byte (allocates a single byte of storage on the stack)
W	Word (allocated on a word boundary)
D	Double-word (allocated on a word boundary)
V	Variable size (allocated on a word boundary)
Q	Quad-word (aligned on a word boundary)
T	Ten-byte word (aligned on a word boundary)
CP	Code pointer (one word for small and compact models)
DP	Data pointer (one word for small and medium models)

The *namelist* parameter specifies the list of the names of the frame variables for the function.

The *size* parameter specifies the size of the variable. It is used with **localV** only.

Comments

B-type variables are not necessarily aligned on word boundaries.

The **localD** macro creates two additional symbols, **OFF_** *name* and **SEG_** *name*. **OFF_** *name* is the offset portion of the parameter; **SEG_** *name* is the segment portion.

Only the name is required when referencing a variable. Write your code like this:

```
mov     al,var1
```

Not like this:

```
mov     al,byte ptr var1[bp]
```

Examples

```
localB <L1,L2,L3>
localW L4
localD <L5>
localV L6,%(size struc)
```

Syntax

cBegin ⟦*procName*⟧

This macro defines the actual entry point for the function **procName**. The macro creates code that sets up the frame and saves registers.

The optional *procName* parameter specifies a function name. If it is given, it must be the same as the name given in the **cProc** macro immediately preceding the **cBegin** macro.

Syntax

cEnd ⟦*procName*⟧

This macro defines the exit point for the function **procName**. The macro creates code that discards the frame, restores registers, and returns to the caller.

The optional *procName* parameter specifies a function name. If it is given, it must be the same as the name given in the **cBegin** macro immediately preceding the **cEnd** macro.

Once a function has been defined using **cProc**, any formal parameters should be declared with the **parm***X* macro and any local variables with the **local***X* macro. The **cBegin** and **cEnd** macros must be used to delineate the code for the function.

The following is an example of a complete function definition:

Example

```
cProc   strcpy,<PUBLIC>,<si,di>
    parmW   dst
    parmW   src
    localW  cnt

cBegin
    cld
    mov     si,src
    mov     di,dest
    push    ds
    pop     es
    xor     cx,cx
    mov     cnt,cx
loop:
    lodsb
    stosb
    inc     cnt
    cmp     al,0
    jnz     loop
    mov     ax,cnt
cEnd
```

7.4.3 Call Macros

The call macros can be used to call **cProc** functions and high-level-language functions. These macros pass arguments according to the calling convention defined by the **?PLM** option.

Syntax

cCall *procName*, ⟦<*argList*>⟧, ⟦<*underscores*>⟧

This macro pushes the arguments in *argList* onto the stack, saves registers (if any), and calls the function *procName*.

The *procName* parameter specifies the name of the function to be called.

The optional *argList* parameter specifies a list of the names of arguments to be passed to the function. This list is not required if the **Arg** macro is used before **cCall**.

The optional *underscores* parameter specifies whether or not an underscore should be added to the beginning of *procName*. If this argument is blank, an underscore is added.

Comments

The arguments of an **Arg** macro are pushed onto the stack before any arguments in the *argList* parameter of a **cCall** macro.

Byte-type parameters are passed as words. There is no sign extension or zeroing of the high-order byte.

Immediate arguments are not supported.

Examples

```
cCall   there,<pExt,ax,bx,pResult>

Arg     pExt
Arg     ax
cCall   there,<bx,pResult>
```

Syntax

Save <*regList*>

This macro directs the next **cCall** macro to save the specified registers on the stack before calling a function, and to restore the registers after the function returns. The macro can be used to save registers that are destroyed by the called function.

The **Save** macro applies to one **cCall** macro only; each new **cCall** must have a corresponding **Save** macro. If two **Save** macros appear before a **cCall**, only the second macro is recognized.

The *regList* parameter specifies a list of registers to be saved.

Examples

```
Save    <cl,bh,si>
Save    <ax>
```

Syntax

Arg <*namelist*>

This macro defines the arguments to be passed to a function by the next **cCall** macro. The arguments are pushed onto the stack in the order given. This order must correspond to the order of the function parameters.

More than one **Arg** macro can be given before each **cCall**. Multiple **Arg** macros have the same effect as a single macro.

The *namelist* parameter specifies a list of argument names to be passed to the function. All names must have been previously defined.

Comments

Byte-type parameters are passed as words. There is no sign extension or zeroing of the high-order byte.

Immediate arguments are not supported.

Examples

```
Arg     var1
Arg     var2
Arg     var3
Arg     <var1,var2,var3>
```

7.4.4 Special-Definition Macros

The special-definition macros inform the **Cmacros** about user-defined variables, function-register use, and register pointers.

Syntax

Def *X* <*namelist*>

This macro registers the name of a user-defined variable with the **Cmacros**. Variables that are not defined using the **static***X*, **global***X*, **extern***X*, **parm***X*, or **local***X* macros cannot be referred to in other macros unless the name is registered, or the variable was defined with the **DW** assembler directive.

The *X* parameter specifies the storage size of the variable. It can be any one of the following:

Type	Description
B	Byte
W	Word
D	Double-word
Q	Quad-word

T	Ten-byte word
CP	Code pointer (one word for small and compact models)
DP	Data pointer (one word for small and medium models)

The *namelist* parameter specifies a list of variable names to be defined.

Example

```
maxSize db      132
        DefB    maxSize
dest    equ     wordptr es:[di]
        DefW    dest
```

Syntax

FarPtr *name, segment, offset*

This macro defines a 32-bit pointer value that can be passed as a single argument in a **cCall** macro. In the **FarPtr** macro, the *segment* and *offset* values do not have to be in registers.

The *name* parameter specifies the name of the pointer to be created.

The *segment* parameter specifies the text that defines the segment portion of the pointer.

The *offset* parameter specifies the text that defines the offset portion of the pointer.

Example

```
FarPtr  destPtr,es,<wordptr 3[si]>
cCall   proc,<destPtr,ax>
```

7.4.5 Error Macros

The error macros allow assertions to be coded into an assembly-language source program. This lets you code optimum instruction sequences for some operations based on the variable allocation or bit position of flag in a word, and assert that the assumptions made are true.

Error macros generate an error message to the console and an error message in the listing. Both the text that caused the error and the result of its evaluation are displayed in the generated error message.

Syntax

errnz <*expression*>

This macro evaluates a given expression. If the result is not zero, an error is displayed.

The *expression* parameter specifies the expression to be evaluated. The angle brackets are required if there are any spaces in the expression.

Examples

```
x       db      ?
y       db      ?

mov     ax, word ptr x
errnz   <(OFFSET y) - (OFFSET x) -1>
```

If, during assembly, *x* and *y* receive anything but sequential storage locations, **errnz** displays an error message.

```
table1  struc
        .
        .
        .
table1len       equ     $-table1
table1  ends

table2  struc
        .
        .
        .
table2len       equ     $-table2
table2  ends

errnz   table1Len-table2Len
```

If, during assembly, the length of two tables is not the same, **errnz** displays an error message.

Syntax

errn$ *label*, [[*bias*]]

This macro subtracts the offset of *label* from the offset of the location counter, then adds *bias* to the result. If this result is not zero, then an error message is displayed.

The *label* parameter specifies a label corresponding to a memory location.

The optional *bias* parameter specifies a signed bias value. A plus or minus sign is required.

Example

```
;       end of previous code
        errn$   function1
function1:
```

If a function that was originally located immediately after another piece of code is ever moved, **errn$** displays an error message.

7.5 Using the Cmacros

This section explains the assembly-language statements generated by some of the **Cmacros** and illustrates their use with an example of a **Cmacros** function called **BITBLT**.

7.5.1 Overriding Types

Parameters and local variables created using the **parm***X* and **local***X* macros actually correspond to expressions of the following form:

```
localB x        ==>         x equ byte ptr [bp+nn]
parmB y         ==>         y equ byte ptr [bp+nn]
```

In this example, the *nn* parameter specifies an offset from the current **bp** register value.

These expressions let you use the names without having to explicitly type in "type ptr" and "[bp+offset]" operators. This means that "x" can be referred to as follows:

```
mov     al,x
```

and that "y" can be referred to as follows:

```
mov     ax,y
```

A problem arises if the type must be overridden. The assembler creates an error message if it encounters the following line:

```
mov     ax,word ptr x
```

This can be solved by enclosing the name in parentheses:

```
mov     ax,word ptr (x)
```

One exception to this pattern is the **localV** macro. The expression generated by this macro does not have a type associated with it. Therefore it can be overridden without the parentheses. For example:

```
localV  horse,10        ==>        horse equ [bp+nn]
```

7.5.2 Symbol Redefinition

Any symbol defined by a **parm***X* macro in one function can be redefined as a parameter in any other function. This allows different functions to refer to the same parameter by the same name, regardless of its location on the stack.

7.5.3 Cmacros: a Sample Function

The following example defines the assembly function **BITBLT**. **BITBLT** is a **FAR** and **PUBLIC** type function. When **BITBLT** is invoked, the **si** and **di** registers are automatically saved, and automatically restored upon exit. Note that the **bp** register is always saved.

BITBLT is passed seven double-word pointers on the stack. Space will be allocated on the stack for eight frame variables (one structure, five bytes, and two words).

The **cBegin** macro defines the start of the actual code. The *pExt* parameter is loaded, and some values are loaded into registers. The **ds** and **si** registers are saved on the following **cCall**.

Another C function, **THERE**, is invoked by the **cCall** macro. Four arguments are passed to **THERE**: *pDestBitmap*, the 32-bit pointer in **ds:si**, register **ax**, and register **bx**. The **cCall** macro places the arguments on the stack in the correct order.

When **THERE** returns, the arguments placed on the stack are automatically removed, and the **ds** and **si** registers are restored.

When **cEnd** is reached, the frame variables are removed, any autosave registers are restored, and a return of the correct type (near or far) is performed.

Example

```
cProc BITBLT,<FAR,PUBLIC>,<si,di>

parmD   pDestBitmap       ;--> to dest bitmap descriptor
parmD   pDestOrg          ;--> to dest origin (a point)
parmD   pSrcBitmap        ;--> to source bitmap descriptor
parmD   pSrcOrg           ;--> to source origin
parmD   pExt              ;--> to rectangle extent
parmD   pRop              ;--> to rasterop descriptor
parmD   pBrush            ;--> to a physical brush

localV  nOps,4            ;# of each operand used

localB  phaseH            ;Horizontal phase (rotate count)
localB  PatRow            ;Current row for patterns [0..7]
localB  direction         ;Increment/decrement flag

localW  startMask         ;mask for first dest byte
localW  lastMask          ;mask for last dest byte

localB  firstFetch        ;Number of first fetches needed
localB  stepDirection     ;Direction of move (left, right)

cBegin

lds     si,pExt
mov     ax,extentX[si]
mov     bx,extentY[si]

RegPtr  dest,ds,si
Save    <ds,si>

cCall   THERE,<pDestBitmap,dest,ax,bx>

mov     extentX[si],cx
mov     extentY[si],dx

cEnd
```

Application Development Tools

Chapter 8
Icon Editor

8.1 Introduction

You can create customized icons, cursors (pointers), and bitmaps for your applications by using the Microsoft Windows Icon Editor. Icon Editor lets you work on a large-scale icon, cursor, or bitmap while it displays the normal-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, Icon Editor lets you create your own unique icons and cursors for your applications.

The following sections explain how to use Icon Editor.

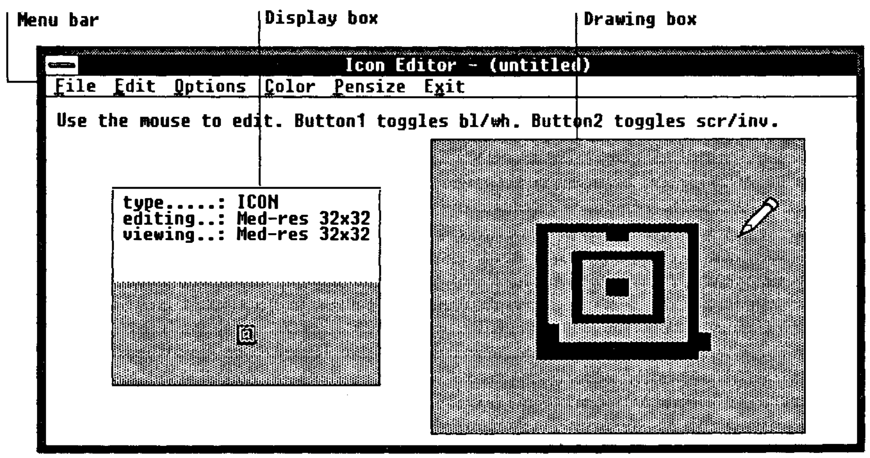
Note

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

8.2 Starting Icon Editor

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

Figure 8.1 shows the Icon Editor window:



Use the mouse to edit. Button1 toggles bl/wh. Button2 toggles scr/inv.

```
type.....: ICON
editing..: Med-res 32x32
viewing..: Med-res 32x32
```

Figure 8.1 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 and clicking the left mouse button. Icon Editor has the following menus:

File
Edit
Options
Color
Pensize
Exit

The display box is the box at the left of the screen. It contains the name of the current editing mode, information on resolution for editing and viewing, and a normal-scale replica of your work. Initially, the drawing type is set to Icon.

The drawing box is the large square box at the right of the screen. This box is the workspace where you will draw 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 a pixel 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.

8.3 Drawing in the Drawing Box

To draw an icon or cursor, you simply move the mouse pointer 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.

Pen size affects the way you color pixels. See Section 8.6 for information on the different pen sizes.

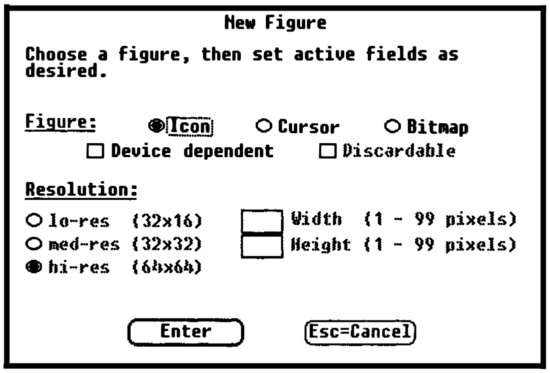
Initially, the pen color is black and the background color is gray. You can change this by using the commands from the Color and Options menus, described in Sections 8.5 and 8.8, respectively. When you are drawing a bitmap, the pen color and background color work slightly differently than in the preceding list; see Section 8.4.3 for information on drawing bitmaps.

You can draw straight lines by using the Options menu and selecting the Draw Straight command.

8.4 Starting a New Drawing

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. The New Figure dialog box will appear. (If you made changes to the current file that you did not save, you will see a dialog box asking you whether you want to save them. If you want to save the changes, choose Yes; otherwise, choose No.) Figure 8.2 shows the New Figure dialog box:



New Figure

Choose a figure, then set active fields as desired.

Figure: [✓ Icon] Cursor Bitmap
☐ Device dependent ☐ Discardable

Resolution:
lo-res (32x16)
med-res (32x32)
[✓ hi-res (64x64)]

Width (1 - 99 pixels)
Height (1 - 99 pixels)

Figure 8.2 The New Figure Dialog Box

8.4.1 Choosing the Drawing Type

Icon Editor has three modes or drawing types: Icon, for drawing and editing icons; Cursor, for drawing and editing cursors; and Bitmap, for drawing and editing patterns. You can choose the drawing type by selecting it from the New Figure dialog box.

To draw an icon, cursor, or bitmap, choose the Icon, Cursor, or Bitmap radio button from the dialog box. Icon Editor will change the label in the display box to reflect the new drawing type. It will also clear the drawing box and re-display the drawing grid.

8.4.2 Choosing Resolution

When you select the drawing type, you will also have the opportunity to select the resolution for editing and viewing Icons and Cursors. The "lo-res" setting refers to CGA-compatible displays, the "med-res" setting corresponds to EGA-compatible displays, and "hi-res" is provided for displays that have a higher resolution than the EGA displays. You cannot select resolution unless you're in device-dependent mode. When you're not in device-dependent mode, the resolution is set at 64×64 for icons and 32×32 for cursors.

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 than this, 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. 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.

8.4.3 Working with Bitmaps

When you select the Bitmap drawing type, Icon Editor changes the label on the display box to Bitmap. If appropriate, it also clears the drawing box and re-displays the drawing grid.

You will need to specify a width and a height for the bitmap. The drawing box can be any size from 1×1 to 99×99 pixels.

When you are drawing bitmaps, the background in the drawing box is set to white and cannot be affected by any choices from the Options menu; the pen color is set to black and cannot be affected by any choices from the Color menu.

You can choose the Discardable checkbox to make your bitmap discardable. This means that if the application needs more memory to run, it can discard the bitmap. Keep in mind that if the application does discard the bitmap, the bitmap will have to be loaded again if it's needed again.

8.4.4 Driver Dependence

If you know precisely which display device the resource that you're creating is going to be used on, then choose the Device dependent checkbox. You can then choose the appropriate resolution, and you can be sure that the icon, cursor, or bitmap that you draw will look the same on the user's screen as it looks in the display box now. However, since most Windows applications will be used on a variety of display devices, you will probably not want to limit yourself by choosing Device dependent.

8.5 Changing the Pen Color

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

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

For icons and cursors, you can use all four colors. For drawing bitmaps, the pen is set to black.

To change the pen color, choose the desired color from the Color menu.

8.6 Changing the Pen Size

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

To change the pen size, choose the size you want from the Pensize menu.

The Small pen size works differently from the Medium, Large, and Extra Large.

8.6.1 Small

When you choose Small, the mouse pointer looks like a pencil when it's in the drawing box. The left mouse button toggles between the pen color and its inverse color; the right mouse button toggles between the screen color and its inverse color.

For instance, if the pen color is black and the screen color is light grey, this is what happens when you're using the Small pen size:

- Pointing to a pixel and clicking the left mouse button turns the pixel black. Clicking the same pixel again turns it white.
- Pointing to a pixel and clicking the right mouse button turns the pixelight gray. Clicking the same pixel again turns it dark gray.

Small is the default pen size. When you start Icon Editor, the pen size is set to Small.

When you are drawing a bitmap, the left and right mouse buttons have the same effect in the Small pen size: either button will turn a black pixel white or a white pixel black.

8.6.2 Medium, Large, and Extra Large

When you choose the Medium, Large, or Extra Large pen size, the pointer looks like a thin brush when it's in the drawing box. The mouse buttons do not toggle. For example, if the pen color is set to black and the background color is set to light gray, this is the effect when you're using the Medium, Large, or Extra Large pen size:

- Pointing to a pixel and clicking the left mouse button turns that pixel and the pixels around it black. Clicking the same pixel again has no effect.
- Pointing to a pixel and clicking the right mouse button turns that pixel and the pixels around it light gray. Clicking the same pixel again has no effect.

When you are drawing a bitmap, the left mouse button always colors the pixel black and the right mouse button always colors the pixel white.

8.7 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 Edit menu. A checkmark appears next to the command name. Information about the current location of the hotspot appears inside 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 row and column information now indicates the location where you placed the hotspot. If you do not set the hotspot, 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 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 Edit menu. When you do so, the checkmark next to Hotspot is removed and the row and column information disappears.

8.8 Changing the Background Color

You can change the background color of the drawing box and the display box by using the Options menu. The Options menu lists four background colors: Black, White, Gray, and Light Gray. To change the background color, choose the desired color command from the Options menu. 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 Inverse 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 white. Since there cannot be any Screen or Inverse Screen pixels in bitmaps, the background color does not matter. The background color is shown only in the display box, outside the area of the bitmap itself; the background color is never part of the bitmap.

8.9 Displaying the Drawing Grid

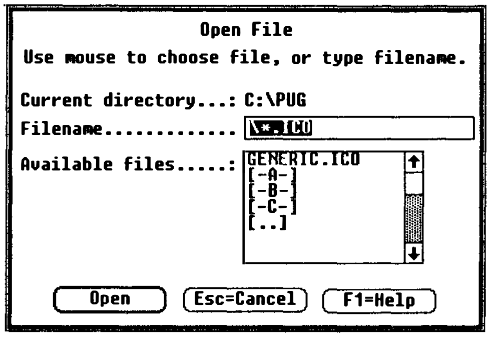
The drawing grid is a grid of lines displayed in the drawing box to help you locate the individual pixels of an icon or cursor.

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

8.10 Opening an Existing Icon, Cursor, or Bitmap File

You can open an existing icon, cursor, or bitmap file by 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 8.3:



Open File
Use mouse to choose file, or type filename.

Current directory...: C:\PUG
Filename............ *.ICO
Available files.....: GENERIC.ICO
[-A-]
[-B-]
[-C-]
[..]

Figure 8.3 Open File Dialog Box

The list box displays files with the extension *.ico*, *.cur*, or *.bmp*, depending on whether 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. (If the file you want to open is in another directory or on another disk, double-click the directory name or the disk name. The filenames in that directory or on that disk that have the appropriate extension will be displayed in the list box, and you can then double-click the name of the file you want.)
   - Select the name in the list box, then choose the Open command or press the ENTER key.
   - Type the name in the text box, then choose the Open command or press the ENTER key. (If the file you want to open is in another directory or on another disk, type the full pathname in the text box.)

   If you want to view a listing of files of a different drawing type, use the wildcard character and the appropriate filename extension in the text box. For instance, to view the listing of bitmap files in the current directory, type **.bmp* in the text box, then press ENTER.

   If you decide not to open a file, choose the Cancel button or press the ESCAPE key.

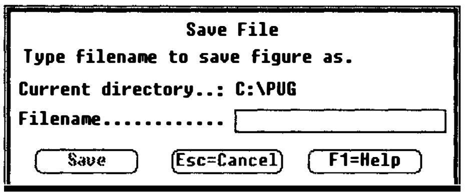
Icon Editor opens the given file. If you have made changes to a file but haven't saved them, Icon Editor displays a dialog box that asks whether you want to save the changes.

8.11 Saving Files

You can save an icon, cursor, or bitmap in a file by using either the Save or the Save As command in the File menu. To save a file under the 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 save it under a new filename, follow these steps:

1. Choose Save As from the File menu. You will see the dialog box shown in Figure 8.4:



```
Save File
Type filename to save figure as.
Current directory..: C:\PUG
Filename............ [            ]
```

Figure 8.4 Save As Dialog Box

2. Type a valid filename in the text box. If you do not type an extension, Icon Editor automatically supplies an extension based on the current drawing type: *.ico* for Icon, *.cur* for Cursor, and *.bmp* for Bitmap.
3. Choose the Save button, or press the ENTER key.

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

8.12 Using the Edit Menu

You can cut selected areas, copy to the Clipboard, and paste from the Clipboard by using the commands in the Edit menu.

If the Clipboard contains data, you can use the Paste command in the Edit menu to paste the material into the drawing box. This allows you to use drawings created with Microsoft Windows Paint, and other Windows applications. The area to be pasted in must be no larger than the current drawing area (in pixels).

You can also select areas of the current drawing and cut or copy them to the Clipboard. To select an area to cut or copy, you must choose either the Select or the Select All command.

The Select command allows you to select an area of the current drawing. To select an area, do the following:

1. Point to a corner of the area you want to select.
2. Drag the mouse pointer diagonally to the opposite corner of the area you want to select, and release the mouse button.

The Select All command selects the entire current drawing for cutting or copying to the Clipboard.

Chapter 9
Font Editor

9.1 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, which contains information about the font, 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

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.*

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

9.2 Starting Font Editor

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

When you start 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 9.3.

9.3 Opening a Font File

Font Editor does not allow you to create fonts from scratch. Instead, you open an existing font file, then make changes to it and, if you wish, save it under a new name as a new font. 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.

   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:

   - Double-click the filename in the list box. (If the file you want to open is in another directory or on another disk, double-click the directory name or the disk name. The filenames in that directory or on that disk that have the *.fnt* extension will be displayed in the list box, and you can then double-click the name of the file you want.)
   - Select the name in the list box, then choose the Open command or press the ENTER key.
   - Type the name in the test box, then choose the Open command or press the ENTER key. (If the file you want to open is in another directory or on another disk, type the full pathname in the text box.)

Note

Font Editor displays an error message if you try to load a file that does not contain a font.

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 9.1:

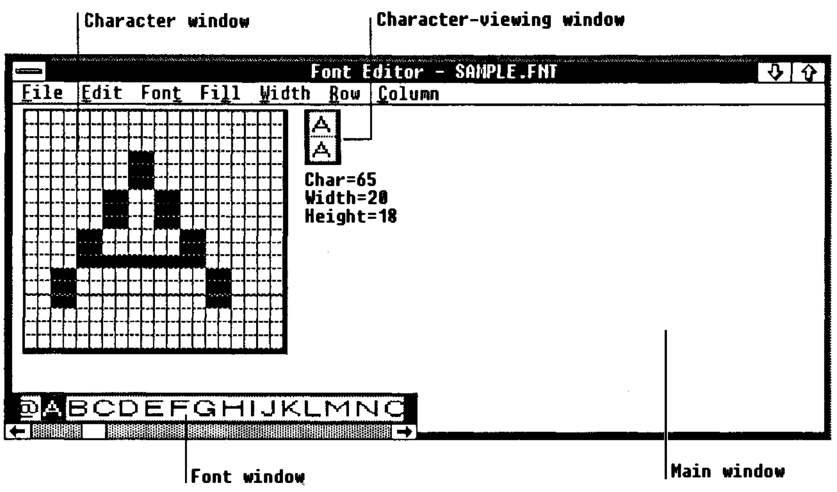


Figure 9.1 Font Editor Window

9.4 Font Editor Features

Font Editor has the following features:

Feature	Description
Main window	This window contains 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 normal-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 normal-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.

9.5 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 mouse pointer into the font-viewing area of the font window.
2. Click the character you want to edit.

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

Warning

When you make a new selection, Font Editor saves the old selection by copying it back to the font buffer. 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. See Section 9.7 for more information on the Refresh command.

While you are editing a font, Font Editor keeps a copy of the font in memory. Changes to individual characters are saved in the buffer, but the font as a whole—including these changes—is not saved until you use the Save command or the Save As command to save it to the font file. See Section 9.23 for more information about saving a font.

9.6 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. Use the mouse button to turn pixels on or off:

- To turn a background pixel on, point to the pixel and click the mouse button.
- To turn several background pixels on, point to a background pixel, then drag the mouse pointer over the pixels you want to change.
- To turn a foreground pixel off, point to the pixel and click the mouse button.
- To turn several foreground pixels off, point to a foreground pixel, then drag the mouse pointer over the pixels you want to change.

9.7 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 in the font buffer. Only the Refresh command cancels changes.

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

9.8 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. 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. If you try to make a character cell wider than the maximum character width, a dialog box will appear, warning you that the maximum character width will be increased.

9.9 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 pixels in the new row are turned on or off in the same pattern as in the selected row.

To copy a row, follow these steps:

1. Choose Add from the Row menu.
2. Move the mouse pointer 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.

9.10 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 mouse pointer 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.

9.11 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 pixels in the new column are turned on or off in the same pattern as those in the selected column.

To copy a column, follow these steps:

1. Choose Add from the Column menu.
2. Move the mouse pointer 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.

9.12 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 mouse pointer 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.

9.13 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 mouse pointer to a pixel in the character window.
3. Press and hold down 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 pointer to another pixel.
5. Release the mouse button.

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 mouse pointer in a horizontal or vertical line, all pixels in the line are cleared.

9.14 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 mouse pointer to a pixel in the character window.
3. Press and hold down 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 pointer to another pixel.
5. Release the mouse button.

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 mouse pointer in a horizontal or vertical line, all pixels in the line become foreground pixels.

9.15 Filling the Character Window with a Hatched Pattern

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 mouse pointer to a pixel in the character window.
3. Press and hold down 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 pointer to another pixel.
5. Release the mouse button.

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 mouse pointer in a horizontal or vertical line, every other pixel in the line becomes a foreground pixel; all others become background pixels.

9.16 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 mouse pointer to a pixel in the character window.
3. Press and hold down 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 pointer to another pixel.
5. Release the mouse button.

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. If you drag the mouse pointer in a horizontal or vertical line, the pixels in the line are inverted.

9.17 Reversing the Character Window

You can reverse the pixels in a block in the character window by using the Left=Right or Top=Bottom commands 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 mouse pointer to a pixel in the character window.
3. Press and hold down 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 pointer to another pixel.
5. Release the mouse button.

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 mouse pointer in a horizontal or vertical line, the pixels in the line are reversed.

9.18 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 mouse pointer to a pixel in the character window.
3. Press and hold down 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 pointer to another pixel.
5. Release the mouse button.

Font Editor uses the anchor point and the second pixel as diagonally opposite corners of the block you want to copy or paste. If you drag the mouse pointer in a horizontal or vertical line, the pixels in the line are copied or pasted.

Be sure that the area in the character window that you want to paste into is the same size as the block on the Clipboard that you want to paste. If you try to paste a block into an area that is larger or smaller than that block, Font Editor will try to stretch or squeeze the block to fit, and the results will be distorted.

Note

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

9.19 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. Font Editor restores the character window to its state before the last change. (If you chose Undo again, it returns the window to its changed state again.)

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 in the buffer (that is, returned to the font).

9.20 Saving Changes to a Character

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

1. Move the mouse pointer into the font-viewing area of the font window.
2. Click the character you are currently editing (it is the highlighted character).

Font Editor saves your selection by copying it back in 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. Font Editor saves the old selection into the font buffer before copying the new selection to the character window. This is useful if you want to continue editing characters in the same font.

9.21 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 possible 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 right.
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 left.

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 from the Font menu. Font Editor displays the Size dialog box, which contains the size options.
2. Select the contents of the text box for the option you want to change. The text box contains a number representing the current choice.
3. Type a new number.
4. Choose the OK button or press the ENTER key.

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

To change the pitch or weight of a font, follow these steps:

1. Choose the appropriate radio button.
2. Choose the OK button or press the ENTER key.

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

9.22 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 contains everything about the font except the bitmap. 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 of an 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 vertical distance (in rows) from the bottom of one character cell to the top of the character cell below it. (The character-viewing window shows two copies of the character, one above the other, so that you can see the effect of leading.)
Internal Leading	This number defines the pixel height of the font's internal leading. Internal leading is the vertical distance (in rows) within a character cell above the top of the tallest letter; only marks such as accents, umlauts, and tildes for capital letters should appear within the space designated as internal leading.

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
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. 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, type a new string or number.
4. Choose the OK button or press the ENTER key.

9.23 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. Font Editor displays a dialog box that contains a text box where you can enter 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. Choose the OK button or press the ENTER key.
   - If the filename you want is shown in the text box, just 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.

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

Note

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

9.24 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.

Chapter 10
Dialog Editor

10.1 Introduction

The Microsoft Windows Dialog Editor lets you design a dialog box 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 use the controls.

This chapter describes how to use 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 use the controls
- Edit an existing dialog box
- Create and modify the include file

Note

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 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 (a binary version of the resource file). You can then use 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 10.10.

10.2 Starting Dialog Editor

To start Dialog Editor, open the MS-DOS Executive window and double-click the filename *dialog.exe.* Windows loads Dialog Editor and displays the window shown in Figure 10.1:

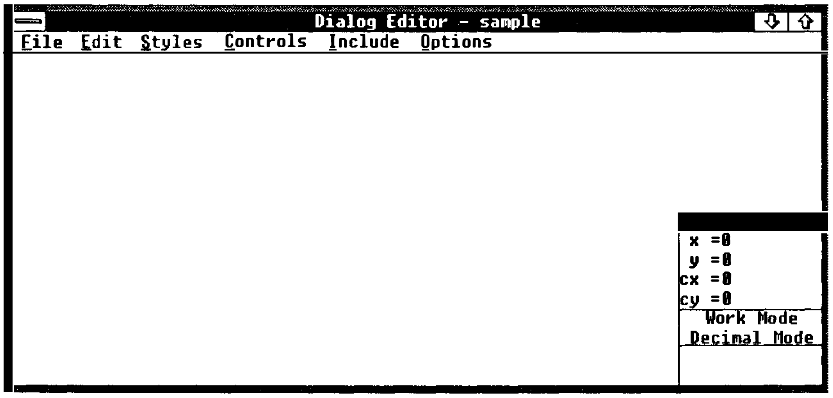


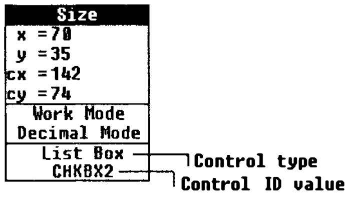
Figure 10.1 Dialog Editor Window

Dialog Editor's menu bar contains the following menus:

Menu	Contents
File	Commands that create, open, and save the files that contain 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	A command that allows you to define the order in which controls are accessed, and a command that allows you to test your dialog box.

10.3 Using the Size Window

When you start 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 10.2 illustrates the Size window:



```
    Size
 x  =70
 y  =35
cx  =142
cy  =74
  Work Mode
 Decimal Mode
   List Box
    CHKBX2
```

Figure 10.2 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 of the width of a character in the system font. One vertical dialog unit is equal to 1/8 of 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 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.

10.4 Creating a Dialog Box

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

10.4.1 Clearing the Display

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

Dialog Editor clears the display, removing any existing dialog box. If you have previously made changes to the existing 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 is initially empty.

10.4.2 Drawing the Border

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

1. Choose the New Dialog command from the Edit menu. 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.

10.4.3 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 10.3:

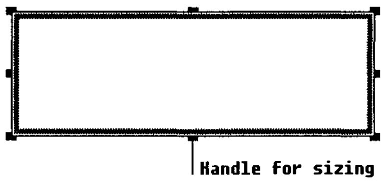


Figure 10.3 Outline of a Dialog Box

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

1. Select the dialog box by clicking inside it. When a dialog box is selected, handles appear on its boundaries.
2. Move the mouse pointer to a handle on the side you want to move. The pointer 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 dialog box will retain its new border. You can size the box in vertical and horizontal directions simultaneously by using a corner handle.

10.5 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 are independent of one another, although two or more often appear next to each other to give the user a choice of selections. Any number of check boxes 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 are typically used in groups to give the user a choice of selections. Only one radio button in a group 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 are usually 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.)

10.5.1 Adding Controls

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

1. Select the control you want from the Controls menu. The mouse pointer changes to a plus sign (+).
2. Position the pointer where you want to place the control.
3. Click the mouse button. The control appears in the dialog box where you placed it. If it has text associated with it, the word "text" is included. To add text, see Section 10.5.2.

10.5.2 Adding Text to Controls

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

1. Select the control by clicking inside it.
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 text box, type the text you want to have appear in the control. You can type more text than will fit in the text box. If you want to edit what you have typed, you can use the DIRECTION keys to move the insertion point, and then add text at the insertion point. 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, which are described in Section 10.6.1.)

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 that the *.rc* file contains the following entry:

```
myicon icon my.ico
```

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

10.5.3 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 pointer 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.

10.5.4 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 down the CONTROL key.
2. While holding down 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 again with the mouse button. You must still be holding down the CONTROL key.)
3. While still holding down the CONTROL key, point to a location inside the group border, but not inside any of the controls' borders, as shown in Figure 10.4:

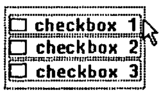


Figure 10.4 Pointer Position for Moving a Group of Controls

4. Press and hold down 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.

Note

You must select *all* the controls you want to move, even if one control encloses another. For instance, to move several radio buttons and the group box that encloses them, you must select each radio button *and* the group box.

10.5.5 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. Handles appear on the boundaries of the control.
2. Move the mouse pointer to a handle. The pointer changes 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 retains the new size.

10.5.6 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 is deleted.

10.6 Changing Control Styles and Memory-Manager Flags

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

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

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

10.6.1 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 10.10.1.)

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, similar to the dialog box shown in Figure 10.5:

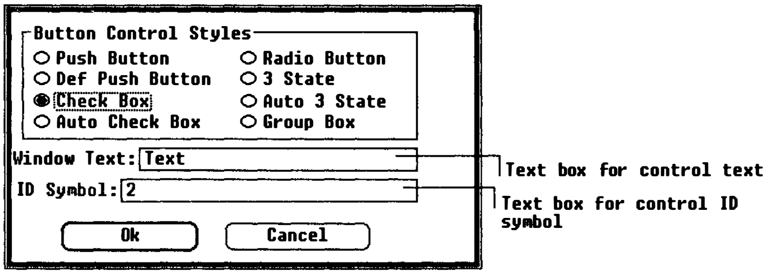

Button Control Styles
○ Push Button ○ Radio Button
○ Def Push Button ○ 3 State
● Check Box ○ Auto 3 State
○ Auto Check Box ○ Group Box
Window Text: Text
ID Symbol: 2

Figure 10.5 Button Control Styles Dialog Box

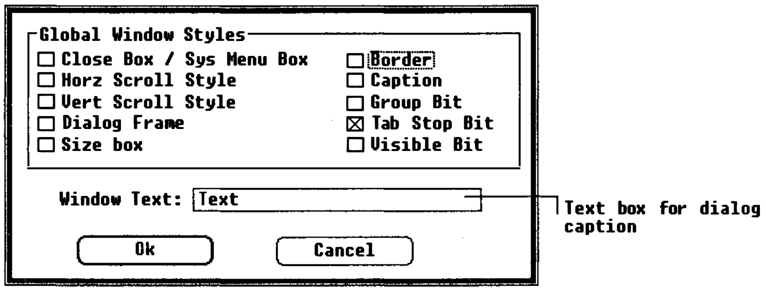
3. Select the desired options. Control-style options are described in Chapter 3, "Resource Compiler: Rc."
4. Choose the OK button or press the ENTER key.

10.6.2 Changing Standard Styles

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

To set global window styles, follow these steps:

1. Select the appropriate control, or the dialog box itself.
2. Choose Standard Styles from the Styles menu. The Standard Styles dialog box will appear, as shown in Figure 10.6:



Global Window Styles

☐ Close Box / Sys Menu Box	☐ Border
☐ Horz Scroll Style	☐ Caption
☐ Vert Scroll Style	☐ Group Bit
☐ Dialog Frame	☒ Tab Stop Bit
☐ Size box	☐ Visible Bit

Window Text: Text

Figure 10.6 Standard Styles Dialog Box

3. Turn on the check boxes for the styles you want to add (or turn off the ones for the styles you want to delete).
4. Add or delete text in the Window Text text box if desired.
5. 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.

10.6.3 Including a System-Menu Box, a Size Box, or Scroll Bars

You can include a system-menu (Control-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 mode-less dialog box must have a system menu so the user can close the box.

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

1. Select the dialog box by clicking a blank area inside it.
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.

10.6.4 Setting Memory-Manager Flags

The Resource Properties command in the Styles menu allows you to set memory-manager flags for your dialog box. 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 or moveable and whether it is discardable.

To set memory-manager flags, follow these steps:

1. Select the dialog box you are working on.
2. Choose the Resource Properties command from the Styles menu. You will see the dialog box shown in Figure 10.7:

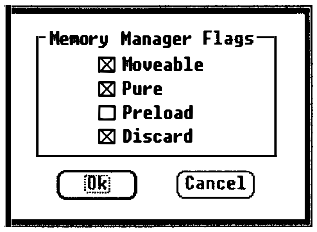

Memory Manager Flags
☒ Moveable
☒ Pure
☐ Preload
☒ Discard

Figure 10.7 Resource Properties Dialog Box

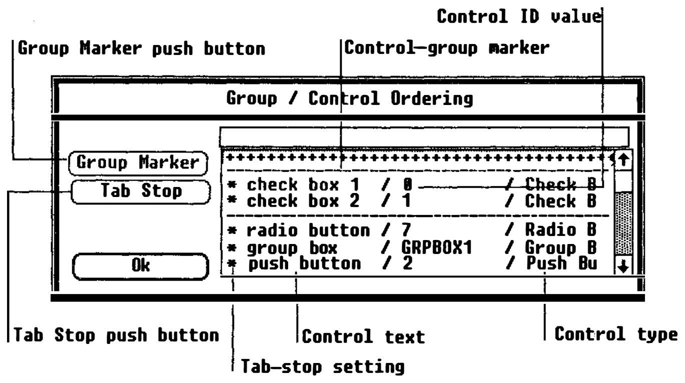
3. Turn on (or off) the check boxes corresponding to the memory-manager flags you want. (These options are described in Chapter 4, "Windows Linker: Link4.")
4. Choose the OK button or press the ENTER key.

10.7 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. You set these options by using 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 control that receives the input focus when the user presses the TAB key).

When you choose the Order Groups command from the Options menu, you will see the Group/Control Ordering dialog box shown in Figure 10.8:



```
++++++++++++++++++++++++++++++++++++++++++++
* check box 1  / 0          / Check B
* check box 2  / 1          / Check B
--------------------------------------------
* radio button / 7          / Radio B
* group box    / GRPBOX1    / Group B
* push button  / 2          / Push Bu
```

Figure 10.8 Group/Control Ordering Dialog Box

10.7.1 Changing the Order of Controls

By default, the controls you place in a dialog box receive the input focus (and thus are selected 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 the controls 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 10.8.
2. In the list box, select the control you want to move.
3. Place the mouse pointer where you want the control to appear in the list box. Notice that as you move it, the pointer 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, click the mouse button.
5. Repeat steps 2 through 4 for any other controls you want to move.

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 click the mouse button. The new selection will replace the old.

10.7.2 Setting a Tab Stop

Tab stops determine where the pointer 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. In the list box, select the control at which you want to place the tab stop.
3. Select the Tab Stop button in the Group/Control Ordering dialog box.
4. Choose the OK button. An asterisk appears next to the control, which indicates that a tab stop has been placed there.

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.

10.7.3 Deleting a Tab Stop

To delete a tab stop, follow these steps:

1. Choose Order Groups from the Options menu.
2. In the list box, select the control that has the tab stop. The Tab Stop button in the Group/Control Ordering dialog box will change to read "Delete Tab."
3. Select the Delete Tab button. The asterisk next to the selected control 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.

10.7.4 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 list box of the Group/Control Ordering dialog box as a horizontal line of hyphens, as shown in the preceding Figure 10.8.) 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. In the list box, 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 by 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.

10.7.5 Deleting a Group Marker

To delete a group marker, follow these steps:

1. Choose Order Groups from the Options menu.
2. In the list box, select the group-marker line. The Group Marker button in the Group/Control Ordering dialog box 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.

10.8 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 application's dialog boxes. If the *.res* file you want is in another directory, type the full pathname of the file in the text box.
2. Open the appropriate *.res* file. You will see a second dialog box, this one 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 third dialog box, this one listing the dialog boxes in the *.res* 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 *.res* file, choose View Dialog from the File menu. You will again see the list of dialog-box names you can choose from.

10.9 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 on 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) on 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 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.

10.10 Using Files with Dialog Editor

The menus and strings that make up the user interface for a Windows application are generally 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 you use Dialog Editor, you modify the *.res* file that contains the binary form of the definitions for the dialog boxes. A typical use of 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 that you created by using Dialog Editor. The include file is described in Section 10.10.1.

10.10.1 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 can then 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 values 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 that 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.*)

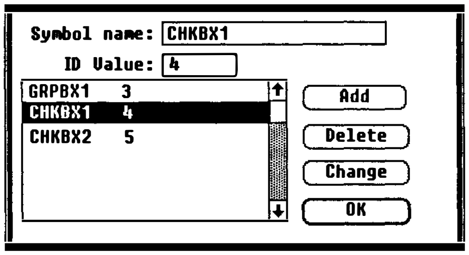
ID Value of 2

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.

10.10.2 Creating an Include File

To create an include file, follow these steps:

1. In the dialog box you are working on, select the control that you want to define in an include file.
2. Choose View Include from the Include menu. You will see a dialog box similar to the one shown in Figure 10.9.


Symbol name: CHKBX1
ID Value: 4

```
GRPBX1   3
CHKBX1   4
CHKBX2   5
```

Figure 10.9 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. Choose the Add button.
6. Choose the OK button or press the ENTER key.
7. Choose Save from the Include menu.

10.10.3 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, similar to the one in Figure 10.9, 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 text box, then select the Change button. To delete the symbol, select the Delete button.
4. Choose the OK button or press the ENTER key.

10.11 Saving a Dialog Box

Once you have created a dialog box or made changes to it, save the new dialog box 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 11
Shaker and Heapwalker

11.1 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 Shaker and Heapwalker.

11.2 Testing Movable Memory: Shaker

The Shaker application lets you see the effect of memory movement on your application. 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. Shaker is useful for making sure that your application locks code and data segments properly when it tries to access them.

To start Shaker, open the MS-DOS Executive window and double-click the filename *shaker.exe*. Windows loads the application and displays the Shaker window.

Table 11.1 shows the Shaker commands:

Table 11.1

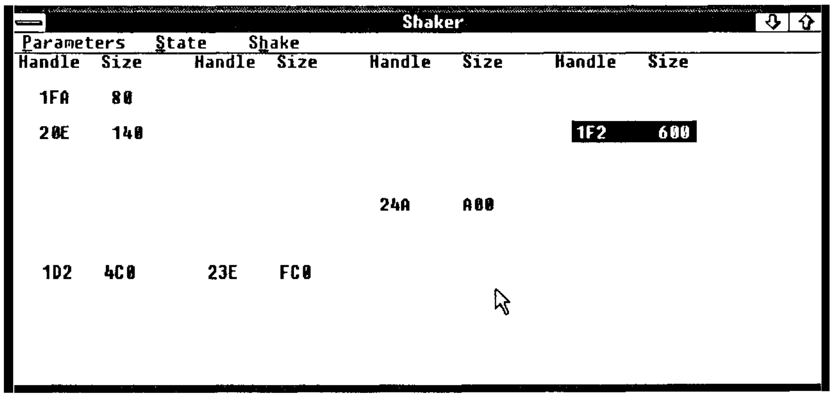
Shaker Commands

Command	**Action**
Parameters Menu	
Allocation Granularity	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, 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	Sets the time interval, in system-timer ticks, between allocations. Shaker allocates a new object after each interval elapses. If the maximum number of objects has been allocated, it reallocates one it has already allocated.

Table 11.1 *(continued)*

Command	Action
Max Objects	Sets the maximum number of objects to be allocated.
State Menu	
On	Displays the object handles and the allocation sizes.
Off	Removes display of object handles and allocation sizes.
Shake Menu	
Start	Starts the allocation.
Stop	Stops the allocation.
Step	Allocates one object and stops. This command can be used when Shaker is otherwise stopped.

Figure 11.1 shows the screen that is displayed when you choose the On command from the State menu:



```
Handle  Size     Handle  Size     Handle  Size     Handle  Size

 1FA     80
 20E     140                                        1F2     600

                                   24A     A00

 1D2     4C0      23E     FC0
```

Figure 11.1 Shaker Window with Show State On

11.3 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 it allocates memory for its own use.

To start Heapwalker, open the MS-DOS Executive window and double-click the filename *heapwalk.exe*. Windows loads the application and displays the Heapwalker window.

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 already been allocated and contain data segments, code segments, resources, etc.

Table 11.2 shows the Heapwalker commands:

Table 11.2

Heapwalker Commands

Command	Action
Walk Menu	
Walk Heap	Displays the current state of memory and identifies each object. Each display line identifies one global object. The display shows the following: • The segment address of the object (actually the segment of the arena header; the object starts one paragraph later) • The size of the object in bytes • The lock count (for example, L2) • Discardable flag, D • The object's owner • The object type (code, data, resource, shared) • Additional information for that object (segment number or name for code, type of resource)
GC(0) and Walk	Performs a global compact, asking for zero bytes, then displays the heap.

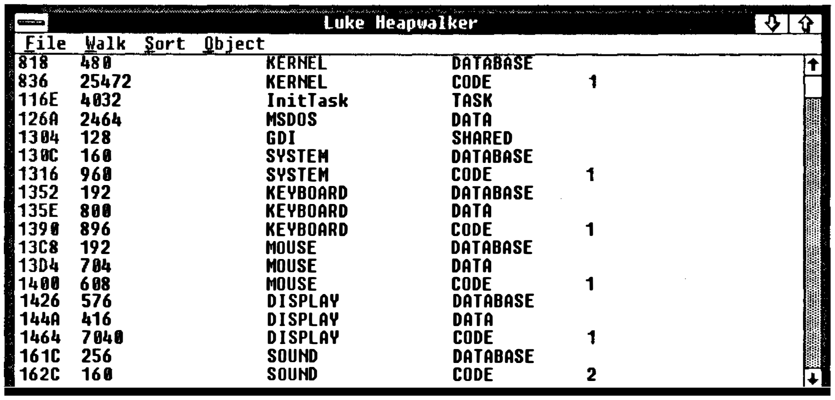
Table 11.2 *(continued)*

Command	Action
GC(–1) and Walk	Allocates all of memory (causing all discardable objects to be thrown out), then displays the heap.
GC(–1) and Hit A:	Used for internal testing.
Allocate all of memory	Allocates all of free memory, which is useful for testing out-of-memory conditions in applications.
Free allocated memory	Frees the memory allocated by the Allocate all of memory command.
Free *x*K of allocated memory	Frees *x* kilobytes of memory. This command applies only to memory allocated by the Allocate all of memory command.
Segmentation Test	Dumps the heap to file *hw6.xx* and does a global compact (–1). This command is available whenever Heapwalker is in the system, even if it is not the active application.
Sort Menu	
Address	Sorts the display by address.
Module	Sorts the display by module name.
Size	Sorts the display by allocation size.
Label Segments	Directs Heapwalker to use *.sym* files in the current directory in order to substitute segment names for segment numbers.
Object Menu	
Show	Displays the contents of the selected object in hexadecimal and ASCII. An object can be selected by clicking the object display.
Show Bits	Displays the bitmap (if any) in the selected GDI object. An object can be selected by clicking the object display.

Table 11.2 *(continued)*

Command	**Action**
LocalWalk	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 Movable) 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. LocalWalk has a File menu with a Save command that saves to a file named *hwl.xx*, where the extension *.xx* is an incremental number appended by the program.
LC(–1) and LocalWalk	Compacts the selected local heap, then displays the heap. LocalWalk has a File menu with a Save command that saves to a file named *hwl.xx*, where the extension *.xx* is an incremental number appended by the program.
GDI LocalWalk	Does a local walk of the GDI local heap and provides expanded information on the objects in the heap. LocalWalk has a File menu with a Save command that saves to a file named *hwl.xx*, where the extension *.xx* is an incremental number appended by the program.
File Menu	
Save	Displays the global heap and writes the results to a file named *hwg.xx*. The extension *.xx* is an incremental number appended by the program.
Exit	Quits the Heapwalker application.
About Heapwalker	Displays information about the current version of Heapwalker.

Figure 11.2 shows the screen that is displayed when you choose the Walk Heap command from the Walk menu:



```
818   480     KERNEL      DATABASE
836   25472   KERNEL      CODE       1
116E  4032    InitTask    TASK
126A  2464    MSDOS       DATA
1304  128     GDI         SHARED
130C  160     SYSTEM      DATABASE
1316  960     SYSTEM      CODE       1
1352  192     KEYBOARD    DATABASE
135E  800     KEYBOARD    DATA
1390  896     KEYBOARD    CODE       1
13C8  192     MOUSE       DATABASE
13D4  704     MOUSE       DATA
1400  608     MOUSE       CODE       1
1426  576     DISPLAY     DATABASE
144A  416     DISPLAY     DATA
1464  7040    DISPLAY     CODE       1
161C  256     SOUND       DATABASE
162C  160     SOUND       CODE       2
```

Figure 11.2 Heapwalker Window after Walk Command

Appendixes

Appendix A 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 A.1 lists most of the diagnostic message numbers and explains the meaning of each message.

Table A.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 A.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 B
C Run-time Functions

Table B.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 that **ds** and **ss** are not equal can be used in Windows libraries.

Table B.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	**dmsbintoieees**	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 B.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 B.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