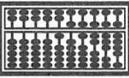
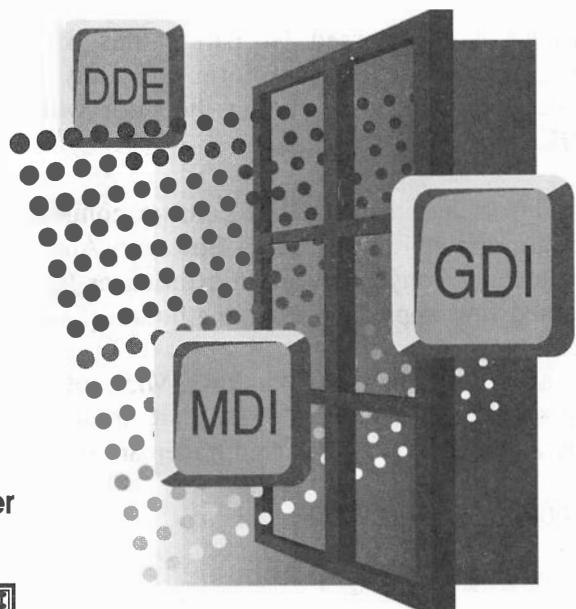


# INTRO TO WINDOWS PROGRAMMING

Juergen Baer and Irene Bauder

**Abacus**   
A Data Becker Book



Printed in U.S.A.

Copyright © 1991 Data Becker GmbH  
Merowingerstrasse 30  
Duesseldorf, Germany

Copyright © 1991 Abacus  
5370 52nd Street SE  
Grand Rapids MI 49512

Edited by: Louise Benzer, Gene Traas, Scott Slaughter, Robbin Markley

This book is copyrighted. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise without the prior written permission of Abacus or Data Becker, GmbH.

Every effort has been made to ensure complete and accurate information concerning the material presented in this book. However, Abacus can neither guarantee nor be held legally responsible for any mistakes in printing or faulty instructions contained in this book. The authors always appreciate receiving notice of any errors or misprints.

DOS, MS-DOS, Microsoft Windows, Microsoft C, Professional Development System, PDS and Software Development Kit (SDK) are trademarks or registered trademarks of Microsoft Corporation. IBM is a registered trademark of International Business Machines Corporation.

ISBN 0-55755-139-1

10 9 8 7 6 5 4 3 2 1

# Contents

<b>Introduction .....</b>	<b>1</b>
Why Windows? .....	1
Getting started.....	1
The companion disk .....	1
What you need to know .....	2
The tools .....	3
What are these tools?.....	3
Installation .....	3
Program layout.....	4
Windows vs. DOS applications .....	4
Overview of Windows 3.....	7
Windows modes.....	7
Real mode .....	7
Standard mode.....	7
Enhanced mode .....	7
Components of a Windows application .....	8
Notation conventions.....	9
The Windows libraries .....	10
Windows functions .....	10
The essential functions .....	11
Description of a window.....	11
Organization .....	12
<b>Your First Windows Application .....</b>	<b>13</b>
Overview .....	13
Block diagram and listing .....	13
Source code: FIRST.C.....	15
Module definition file: FIRST.DEF .....	17
Compiling and linking .....	18
How FIRST.EXE works .....	19
Header file .....	19
WinMain .....	20
Initialization.....	22
Window creation .....	24
Child windows .....	25
Popup windows .....	26
MSG structure.....	27
Message queues .....	27

Focus .....	29
Message loop.....	30
Window functions.....	31
Default window functions .....	32
More about messages.....	32
Exiting an application.....	34
The module definition file .....	35
Module definition file keywords.....	35
Dynamic Link Library (DLL) .....	38
Example of IMPORTS and EXPORTS.....	38
Example of dynamic linking .....	40
<b>Keyboard and Mouse Input .....</b>	<b>43</b>
Input messages.....	43
Keyboard input.....	43
Keyboard messages.....	44
Character messages.....	45
Keyboard example .....	49
Source code: KEYDEMO.C .....	49
Module definition file: KEYDEMO.DEF .....	53
How KEYDEMO.EXE works .....	54
Mouse input .....	54
Mouse messages.....	55
wParam:.....	56
lParam:.....	56
Mouse example .....	57
Source code: MOUSE_BT.C .....	58
Module definition file: MOUSE_BT.DEF .....	61
How MOUSE_BT.EXE works .....	62
Timer input.....	63
Timer messages.....	63
Timer example .....	64
Source code: TIMER.C .....	65
Module definition file: TIMER.DEF .....	68
How TIMER.EXE works .....	68
<b>Output.....</b>	<b>71</b>
Overview .....	71
Device context.....	71
Device context attributes .....	72
Device context types .....	73

Display context types.....	74
The WM_PAINT message .....	76
Origin .....	76
Processing messages.....	77
Paintstruct:.....	78
Output functions .....	79
Text functions .....	79
Example of the DrawText function.....	83
Source code: TEXT.C .....	84
Module definition file: TEXT.DEF.....	86
How TEXT.EXE works .....	87
Example of basic graphic functions .....	89
Source code: LINES.C .....	91
Module definition file: LINES.DEF .....	94
How LINES.EXE works .....	95
Example of ellipse and polygon output.....	98
Source code: GRAPHICS.C.....	99
Module definition file: GRAPHICS.DEF .....	102
How GRAPHICS.EXE works.....	102
Drawing tools.....	103
Using the tools .....	104
Pen .....	104
Brush .....	106
Example of Pen and Brush tools.....	108
Source code: TOOLS.C .....	109
Module definition file: TOOLS.DEF.....	113
How TOOLS.EXE works .....	114
Font .....	115
LOGFONT structure:.....	118
Example of font tools.....	120
Source code: FONT.C .....	121
Module definition file: FONT.DEF.....	126
How FONT.EXE works .....	127
<b>Resources .....</b>	<b>129</b>
Overview.....	129
Icons .....	131
Example of icon access from an RC file.....	134
Source code: ICON1.C .....	135
Module definition file: ICON1.DEF.....	137
Resource script: ICON1.RC .....	137
How ICON1.EXE works .....	138

Example of dynamic icon access.....	139
Source code: ICON2.C .....	139
Module definition file: ICON2.DEF.....	142
How ICON2.EXE works .....	143
<b>Cursor</b> .....	143
Example using a self-defined cursor.....	145
Source code: CURSOR.C .....	146
Module definition file: CURSOR.DEF .....	149
Resource script: CURSOR.RC .....	150
How CURSOR.EXE works.....	150
<b>Other single-line statements</b> .....	151
BITMAP .....	151
FONT .....	151
<b>STRINGTABLE</b> .....	152
<b>Menus</b> .....	153
Menu definition in resource scripts .....	153
Linking to the source text.....	156
Evoked messages .....	156
Menu changes.....	157
Changes to menu item's status: .....	158
Changes to menu item's presence: .....	158
Changes to the entire menu: .....	158
Information function:.....	158
Floating pop-up menus .....	159
Defining your own checkmarks.....	159
Menu example.....	160
Source code: MENU.C .....	161
Module definition file: MENU.DEF.....	167
Resource script: MENU.RC .....	167
Header file: MENU.H.....	169
How MENU.EXE works .....	170
<b>Keyboard accelerators</b> .....	173
<b>Dialog boxes</b> .....	174
Controls .....	174
Control classes:.....	175
Notification codes: .....	176
Creating dialog boxes.....	178
Types .....	181
Dialog routine.....	181
Calling the dialog box .....	183

Example of a modal dialog box.....	185
Source code: DIALOGBX.C .....	186
Module definition file: DIALOGBX.DEF.....	192
Resource script: DIALOGBX.RC .....	192
Header file: DIALOGBX.H.....	192
Dialog box file: DIALOGBX.DLG .....	193
How DIALOGBX.EXE works .....	194
Message boxes .....	197
<b>Mapping Mode .....</b>	<b>199</b>
Overview .....	199
<b>Mapping mode types .....</b>	<b>200</b>
Device-dependent mode.....	200
Metric modes .....	201
Custom modes.....	203
<b>Example of custom modes .....</b>	<b>205</b>
Source code: MAPPING.C.....	205
Module definition file: MAPPING.DEF.....	210
Resource script: MAPPING.RC.....	210
Header file: MAPPING.H .....	211
How MAPPING.EXE works.....	211
<b>Bitmaps .....</b>	<b>215</b>
Overview .....	215
<b>Device-dependent bitmaps.....</b>	<b>215</b>
Creating a bitmap .....	215
Creating a bitmap as a resource:.....	216
Bitmap appearance .....	217
Bitmap coding .....	217
Creating bitmaps with GDI functions .....	218
Creating bitmaps with a bit array .....	218
Output functions .....	218
More about bitmaps .....	223
<b>Example of a device-dependent bitmap.....</b>	<b>225</b>
Source code: BIT1.C .....	226
Module definition file: BIT1.DEF.....	229
Resource script: BIT1.RC .....	230
How BIT1.EXE works .....	230
<b>Device-independent bitmaps (DIB) .....</b>	<b>232</b>
DIB structure.....	232
DIB functions.....	235

Example of DIB functions .....	238
Source code: BIT2.C .....	239
Module definition file: BIT2.DEF.....	243
Resource script: BIT2.RC .....	243
How BIT2.EXE works .....	244
<b>Scroll Bars.....</b>	<b>247</b>
Overview.....	247
Working with scroll bars.....	247
Defining scroll bars .....	247
Scroll range and scroll bar position .....	248
Scroll bar messages.....	249
Keyboard support .....	251
Scrolling.....	252
Example of scroll bars .....	254
Source code: SCROLL.C .....	255
Module definition file: SCROLL.DEF.....	260
Resource script: SCROLL.RC .....	260
How SCROLL.EXE works .....	261
<b>The Clipboard.....</b>	<b>265</b>
Overview.....	265
Text format.....	266
Copying text to the clipboard.....	266
Copying text to global memory: .....	266
Open the clipboard:.....	266
Empty the clipboard's contents:.....	266
Pass handle to global memory range: .....	266
Close the clipboard: .....	267
Obtaining text from the clipboard .....	267
Open the clipboard:.....	267
Get handle to global memory area from clipboard: .....	268
Copy data to a global memory area:.....	268
Close the clipboard: .....	268
Example of text format.....	269
Source code: CLIP.C.....	270
Module definition file: CLIP.DEF .....	275
Resource script: CLIP.RC.....	275
Header file: CLIP.H .....	275
How CLIP.EXE works .....	276

<b>Bitmap format.....</b>	<b>278</b>
Writing a bitmap to the clipboard .....	278
Obtaining a bitmap from the clipboard.....	279
Open the clipboard:.....	279
Get handle to global memory area from clipboard: .....	279
Display bitmap on the screen:.....	279
Close clipboard:.....	279
Example of bitmap format.....	280
Source code: CLIPBIT.C .....	281
Module definition file: CLIPBIT.DEF .....	289
Resource script: CLIPBIT.RC .....	290
Header file: CLIPBIT.H.....	290
How CLIPBIT.EXE works.....	291
<b>Additional information about formats .....</b>	<b>292</b>
Multiple formats in the clipboard .....	292
Delayed rendering .....	293
Your own data formats .....	294
<b>Clipboard viewer .....</b>	<b>295</b>
Old status:.....	297
New status:.....	297
Example of a clipboard viewer .....	297
Source code: VIEWER.C.....	298
Module definition file: VIEWER.DEF .....	301
How VIEWER.EXE works.....	302
<b>File Management.....</b>	<b>303</b>
<b>Overview.....</b>	<b>303</b>
<b>MS-DOS files .....</b>	<b>303</b>
OpenFile function .....	304
OFSTRUCT: .....	305
Closing a file .....	306
Reading a file.....	306
Writing to a file.....	307
Setting the file pointer.....	307
Example of file access.....	308
Source code: FILE.C.....	309
Module definition file: FILE.DEF .....	317
Resource script: FILE.RC.....	317
Header file: FILE.H .....	318
Dialog box file: ENTRIES.DLG.....	318
Dialog box file: OPENNEW.DLG .....	319
How FILE.EXE works .....	319

<b>Initialization Files .....</b>	<b>321</b>
Standard initialization files .....	322
SYSTEM.INI configuration ranges:.....	325
Creating your own initialization files.....	325
Example of initialization files .....	326
Source code: FINIT.C .....	327
Module definition file: FINIT.DEF.....	334
Resource script: FINIT.RC .....	334
Header file: FINIT.H.....	334
Dialog box file: MEALS.DLG .....	335
How FINIT.EXE works .....	336
<b>Dynamic Link Libraries.....</b>	<b>339</b>
Introduction .....	339
FAR functions .....	340
Basics.....	340
Prolog and Epilog .....	340
Callback functions .....	342
Restrictions .....	344
Creating a DLL .....	347
Source code .....	347
Resources .....	348
Wep .....	348
Module definition file .....	349
Compiling and linking DLLs .....	350
Adding the DLL to an application .....	351
DLL examples.....	353
Example of DLL resource access .....	353
DLL source code: DLLRC.C.....	354
DLL module definition file: DLLRC.DEF.....	354
DLL resource script: DLLRC.RC.....	355
Header file: DLL.H.....	356
Application source code: FIRSTDRC.C.....	356
Application module definition file: FIRSTDRC.DEF .....	360
How FIRSTDRC.EXE and DLLRC.EXE work.....	360
Example of DLL function access .....	362
DLL source code: DLLFUNC.C.....	363
DLL module definition file: DLLFUNC.DEF.....	364
DLL header file: DLLFUNC.H.....	365
Application source code: FIRSTDFC.C .....	365
Application module definition file: FIRSTDFC.DEF .....	369

How FIRSTDFC.EXE and DLLFUNC.DLL work .....	370
<b>The Help System.....</b>	<b>373</b>
Overview.....	373
Help application.....	374
File .....	374
Edit .....	374
Bookmark.....	375
Help .....	375
Creating a Help system.....	375
Planning .....	376
Help topic files .....	378
Context string.....	378
Title .....	379
Keyword.....	380
Browse sequence number.....	380
Cross reference .....	382
Definitions .....	383
BUILDTAG .....	384
Help Project file .....	385
[FILES] .....	386
[OPTIONS] .....	386
[BUILDTAGS] .....	388
[MAP].....	388
[ALIAS].....	389
The HC.EXE help compiler .....	390
Programming the application.....	391
Example of the Help system.....	392
Source code: HELP.C.....	393
Module definition file: HELP.DEF .....	396
Resource script: HELP.RC.....	397
Header file: HELP.H .....	397
Help project file: HELP.HPJ .....	402
<b>The Multiple Document Interface.....</b>	<b>403</b>
General information.....	403
Structure of an MDI application .....	403
Additions and changes .....	404
Message Loop .....	404
Frame window .....	405
MDI client window .....	406
CLIENTCREATESTRUCT .....	407

MDI child window.....	407
MDICREATESTRUCT .....	410
Implicit MDI child window styles .....	410
Optional MDI child window styles.....	411
WM_MDIACTIVATE .....	412
MDI example .....	413
Source code: MDI.C.....	414
Module definition file: MDI.DEF .....	423
Resource script: MDI.RC.....	424
Header file: MDI.H .....	425
How MDI.EXE works.....	425
Closing all windows and icons .....	427
<b>Index .....</b>	<b>431</b>

# Introduction

In this chapter we'll give you a brief description of what you need to get started in programming Windows applications. This information will include what you should know, what software you should have, and how a Windows application is constructed.

## Why Windows?

What makes Windows so attractive to the software developer? The graphic user interface provided by Windows allows the developer to quickly and easily create user-friendly, mouse based applications. The Windows settings specified by Setup and the Windows Control Panel save the programmer the trouble of writing separate installation programs. Instead, Windows handles the configuration and the application takes advantage of the existing configuration.

Microsoft Windows 3 is considered the standard for graphic user interfaces. More people are writing applications compatible with this interface.

## Getting started

### The Companion Disk

The Companion Disk for Intro to Windows Programming contains the source code for all program examples listed in the book. The book will give a full description of each program along with instructions on its use. This disk is not intended as a stand alone tutor; please read the section of the book describing a program before attempting to use it.

The examples were written using Microsoft C Version 6 and the Microsoft Software Development Kit (SDK). Windows 3.x is required to run the compiled programs.

The programs on the companion disk are in a compressed format, and must be installed to your hard disk before they can be used. See the

README.TXT file on the companion diskette for the complete installation instructions.

## **What you need to know**

Let's start with what experience you'll need for reading this book. Although you can read this book without the following knowledge, we recommend that you do further studying, if possible, to avoid confusion.

We assume that you:

- Have experience with developing applications in the C language. If you haven't programmed in the C language, there are many books available on the subject.
- Have experience operating MS-DOS based PCs.
- Have written applications to run under MS-DOS.
- Have experience with Microsoft Windows.

## **The tools**

You'll need the following software to write a Windows application:

- A C compiler. If you're an experienced C programmer, you probably have access to a compiler already. The source codes in this book were written in the C language using the Microsoft C Compiler Version 6.00 and the Professional Development System (PDS). This software is available from Microsoft Corporation.
- The Microsoft Windows Software Development Kit (SDK), also available from Microsoft Corporation. We used Version 3.0. This contains many tools for writing Windows applications, including a paint program for creating your own icons.

## What are these tools?

The C compiler includes:

- A compiler and linker (CL.EXE and LINK.EXE) for compiling source code and turning that source code into executable program code.
- Header files such as DOS.H and CONIO.H. Header files contain definitions, macros and other important programming data.
- Standard library files for handling different memory models: SLIBCE.LIB, MLIBCE.LIB and LLIBCE.LIB (Small, Medium and Large).

The Software Development Kit (SDK) makes a program into a Windows application. Its features include:

- A linker (LINK.EXE or LINK4.EXE) and resource compiler (RC.EXE).
- The CodeView debugging application (CVW.EXE) for finding errors in applications.
- Windows-specific header files and library files.
- A font editor (FONTEdit.EXE) and an icon paint program (SDKPAINT.EXE). SDKPAINT is similar to Microsoft Windows Paintbrush, except that SDKPAINT allows you to create cursors, bitmaps and icons for inclusion in Windows applications.

## Installation

We suggest that you install the Microsoft C compiler first, then install the SDK in a separate directory. For example, we installed Microsoft C in the D:\C600 directory, and the SDK in the D:\WINDEV directory. We then changed the AUTOEXEC.BAT file to reflect the following:

```
PATH=C:\DOS;C:\WINDOWS;D:\C600\BIN;D:\C600\BINB;D:\WINDEV;D:\WINDEV\INCLUDE;
      D:\WINDEV\INCLUDE\SYS
SET LIB=D:\WINDEV\LIB;D:\C600\LIB
SET INCLUDE=D:\WINDEV\INCLUDE;D:\C600\INCLUDE
SET HELPFILES=D:\C600\HELP\*.HLP
SET INIT=D:\C600\INIT
```

## Program layout

The program listings in this book will appear in **courier** font. The program listing's left margin is aligned with the chapter and section titles. Here's an example of program text as you'll find it in this book:

```
*****
```

### E X A M P L E

This is an example of left flush program code as it appears in the Abacus book

W I N D O W S   P R O G R A M   D E V E L O P M E N T

```
*****
```

Some program lines go beyond 80 characters in line length and have been split to the next line and indented. See the companion diskette for this book for the complete source code listings.

## Windows vs. DOS applications

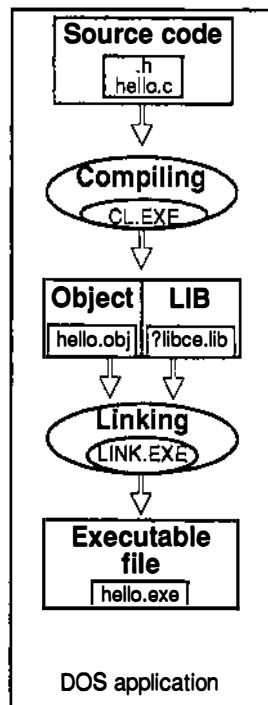
Let's look at how developing Windows applications differs from developing standard DOS based applications. Developing a DOS based application begins with the source code (written in the C language, for this example).

Once the developer writes the source code and ensures that the code is error-free, the code can be run through a compiler (CL.EXE). The source code (we'll call it HELLO.C in this example) references header files (files with .H extensions) during compilation.

These header files (sometimes called *include* files because the C language #include statement requires these files in the source code) contain information needed by the completed program (e.g., for screen display).

The compiler generates an object file (in this case, a file named HELLO.OBJ). One step remains: The .OBJ file and a library file (?LIBCEW.LIB) must be linked by LINK.EXE to form an executable file. The final result is the file HELLO.EXE.

The following illustration shows how a DOS application is assembled, from C source code to executable file:



Generating a Windows application from scratch uses the same steps, plus some additional programming and preparation. The application undergoes the same process (compiling and linking), with a few major differences:

- 1) An additional include file named WINDOWS.H is referenced with the other include files and the source code. WINDOWS.H contains Windows-specific type declarations, macros and other information.
- 2) Instead of the ?LIBCE.LIB files used by C, Windows-specific libraries are used by the linker (?LIBCEW.LIB and LIBW.LIB).

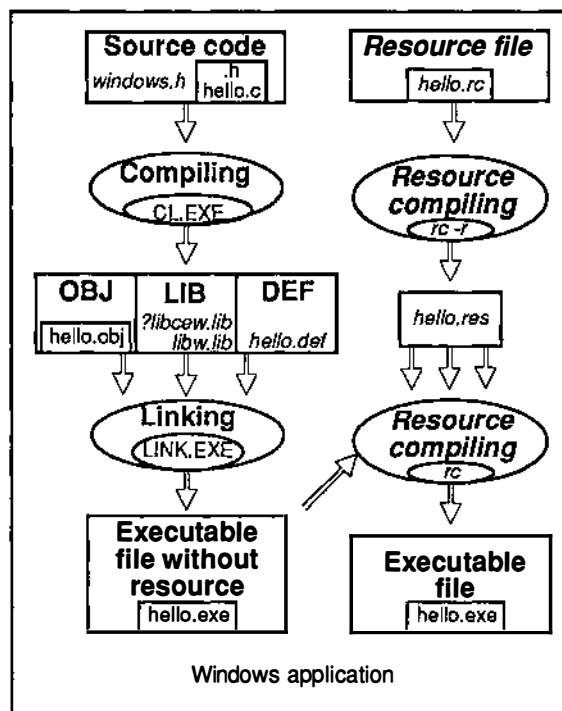
- 3) A module definition file (indicated by a .DEF extension) is linked to the object file and library. The module definition file defines name, description, data types and other materials needed.

The linked file still isn't ready for operation under Windows. It requires resources. Bitmaps, strings, icons and cursors are examples of Windows resources.

Resource scripts (identified by an .RC extension) are compiled using the resource compiler RC.EXE. The resulting file is added to the linked .EXE file by RC.EXE. The final result is an application compatible with Windows 3.

If you don't compile resources, simpler applications will run under Windows 3, but you'll encounter an application compatibility error when you start the application.

The following illustration shows how a Windows application is generated, from source code to executable file:



# Overview of Windows 3

Windows 3 was introduced in the spring of 1990. The Windows 3.0 graphic user interface is considered a user environment of DOS that is capable of multitasking.

## Windows modes

Windows 3 has three different modes: Real mode, standard mode, and enhanced mode. These modes are defined as follows:

### Real mode

Real mode is very similar to the segmented addressing used by MS-DOS. This mode operates in conventional memory only (up to 1 Meg) and requires a minimum of 640K of free memory to start successfully. Real mode has the advantage of allowing you to run applications written for older versions of Windows in Windows 3.

### Standard mode

Standard mode exceeds the 1 Meg limit set by MS-DOS and real mode. The amount of extended memory is added to the free conventional memory and this total area is considered contiguous memory. In addition, a block from the High Memory Area (HMA) is added to the extended and conventional memory.

MS-DOS based programs must execute in conventional memory, just like Windows running in real mode.

### Enhanced mode

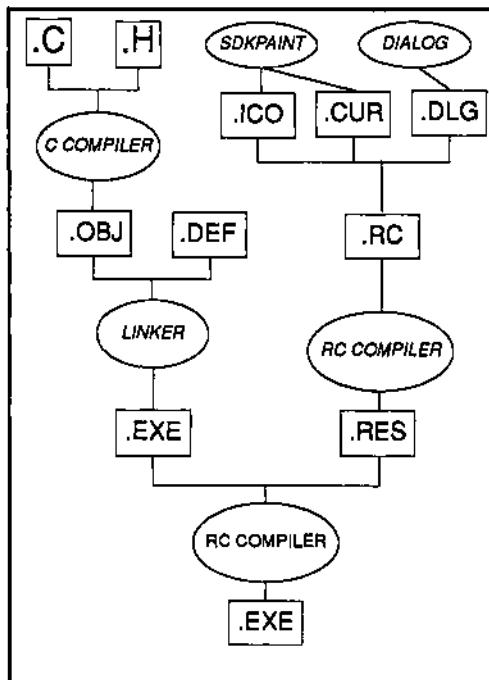
In the Enhanced mode, which can only be used on the 80386 or 80486 processors, virtual DOS devices are created for the individual DOS programs. The virtual memory is available to all Windows applications. This means that the "paged" external memory is also available.

The Windows Virtual Memory Manager (VMM) is responsible for storing and recalling recently unused pages to and from the hard disk.

Each of these pages is 4K in size and can be stored in either a temporary or permanent swap file. Windows names the temporary file WIN386.SWP. The permanent file can be created with the SWAPFILE utility in the Real mode. This file is automatically named 386SPART.PAR and contains the hidden and system attributes.

## Components of a Windows application

Most Windows applications are written in C, as are all the applications listed in this book. We mentioned earlier that to create an executable application, other files are needed in addition to the source file. These additional files are displayed in the following illustration. We've used only the most common files throughout this book.



Some Windows applications involve several files in compiling and linking. As we discuss each example, we'll list the files needed.

# Notation conventions

Many applications use Hungarian notation, which specifies that every variable name should start with one or more letters that indicate the data type of this variable. Often the handle for a window is called hWnd; the "h" represents "handle". This makes programs easier to read, once you become familiar with the notation. Here's a list of Hungarian notation prefixes and their meanings:

<b>Prefix</b>	<b>Meaning</b>
b	BOOL (int)
by	BYTE (unsigned char)
c	char
cx,cy	short (count - x length or y length)
dw	unsigned long (DWORD)
fn	function
i	int
l	long
n	int or short
s	string
sz	string terminated with null byte
w	unsigned int (WORD)
xy	short (x coordinate or y coordinate)

Windows has many of its own types in addition to standard C data types. The WINDOWS.H header file defines these types, which are as follows:

<b>Type</b>	<b>Meaning</b>
BOOL	Boolean 16 bit value
BYTE	Unsigned 8 bit value
char	Signed 8 bit value (can also consist of ASCII characters)
DWORD	Unsigned 32 bit value
far or FAR	Instructs compiler to use a long pointer (32 bit address)
HANDLE	General handle (16 bit value)
HDC	Handle to a device context (16 bit value)
HWND	Handle to a window (16 bit value)
int	Signed 16 bit value
long or LONG	Signed 32 bit value
LPINT	Long pointer to type int
LPSTR	Long pointer to character string
near or NEAR	Instructs compiler to use a short pointer
PINT	Short pointer to type int
PSTR	Short pointer to character string
short	Signed 16 bit value
WORD	Unsigned 16 bit value
void or VOID	Indicates that function does not return a value

## The Windows libraries

The definitions for all Windows functions are stored in libraries. Unlike runtime libraries as in the C language, these libraries are Dynamic Link Libraries (DLLs), references to which are linked to the application when the application is loaded. Since the libraries themselves aren't linked to the application, Dynamic Link Libraries (DLLs) can be updated without having to rewrite the applications itself.

Windows itself uses the following Dynamic Link Libraries:

- KERNEL.EXE is responsible for the memory and resource management and for multitasking.
- USER.EXE regulates the window management and handles input.
- GDI.EXE provides the graphic interface and handles all output.

We'll discuss Dynamic Link Libraries (DLLs) in detail later.

## Windows functions

All the functions that are provided with the Windows SDK can be divided into three groups: Window Manager interface functions, Graphics Device interface functions, and System Services interface functions.

The Window Manager interface consists of 18 subgroups, such as message functions, menu functions, and scrolling functions.

The Graphics Device Interface (GDI) consists of 17 subgroups, such as device context functions, text functions, and mapping functions.

The System Services interface consists of 15 subgroups, such as Memory Manager functions, Resource Manager functions, and file I/O functions.

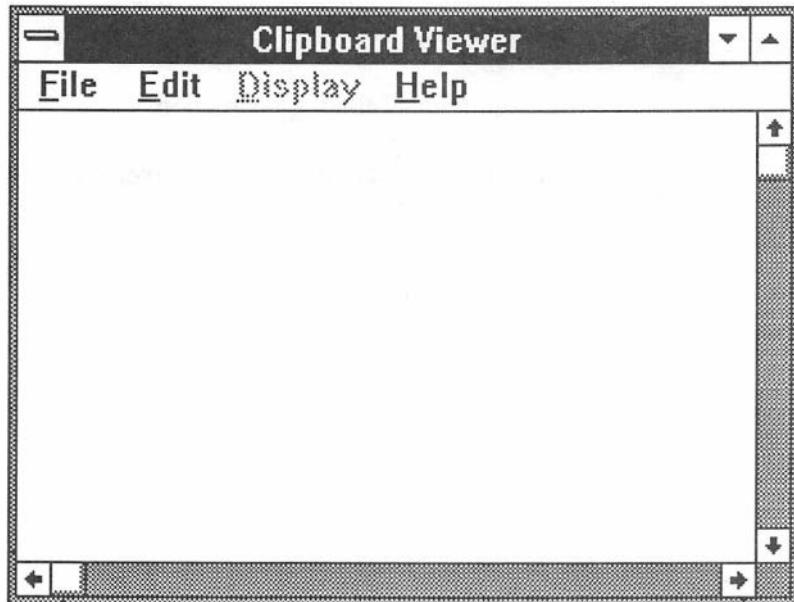
You'll encounter many functions from these groups in the next chapters.

## The essential functions

A Windows application's source code must have two basic functions:

- The WinMain function. This replaces the main() function found in traditional C language programs. The WinMain function represents the "point of entry" for program execution, providing basic operations and parameters.
- The main window function. This function controls basic window display and passes any messages to the window (more on messages in the next chapter). This function can have any name assigned to it—we assign names that make the function obvious (e.g., MainWndProc).

## Description of a window



The client area is important to the programmer because it is his/her work area. The programmer determines the content that appears on the screen. The other components are handled by the system itself. For

example, the system ensures that the System menu always appears in the upper-left corner of each window.

## Organization

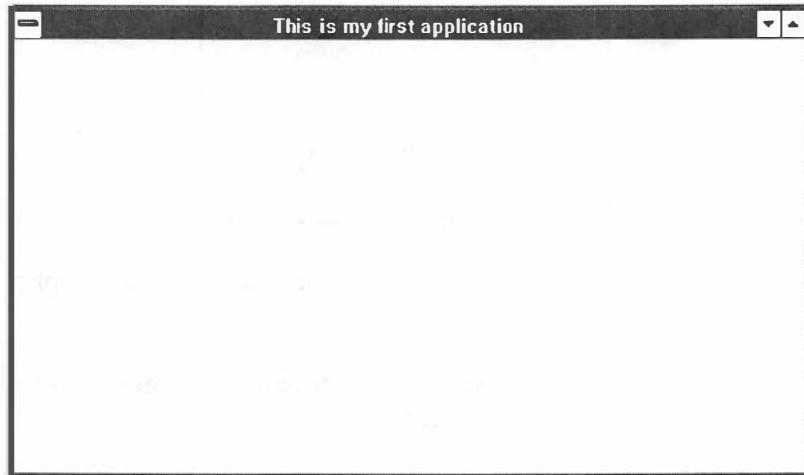
Each time we discuss an application in this book, we'll give you information in the following order:

- Documentation of functions and code used in the application.
- Brief description of the application.
- Brief descriptions of new messages, new functions, and new structures as they appear in the application.
- Commented source code of the application itself.
- Module definition file source code and other source codes where applicable.
- Compiling and linking instructions.
- How the application works and other information.

# Your First Windows Application

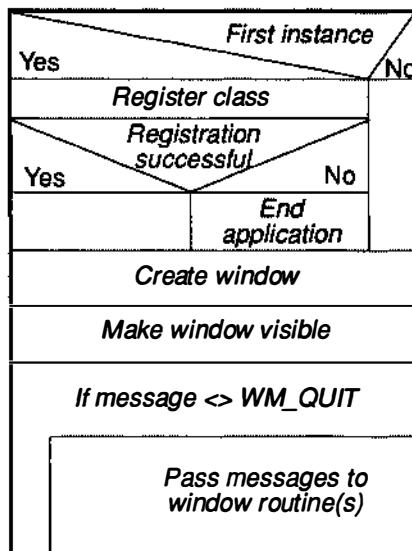
## Overview

In this chapter we'll explain the important parts of a Windows application. We'll talk about initialization, the message loop, the main window function, and the module definition file.



## Block diagram and listing

We'll use the following block diagram of the WinMain function as a basis for our discussion of the functions and related concepts:



As we mentioned earlier, a Windows application must have the following two components:

- The WinMain function is the beginning of the application and the main procedure.
- A main window function. In the FIRST.C source code, the FirstWndProc is the main window function. This function processes all messages (more on messages later in this chapter).

The source code listed below displays a window when compiled and linked. This window's title bar contains the words, "This is my first application."

New messages	Brief description
WM_DESTROY	Sent during the destruction (removal) of the window
New functions	Brief description
CreateWindow	Creates a window
DefWindowProc	Default window procedure
DispatchMessage	Passes messages to WndProc
GetMessage	Gets messages from the queue
PostQuitMessage	Creates a WM_Quit message
RegisterClass	Registers a class

---

ShowWindow	Shows the window
TranslateMessage	Translates keyboard messages
UpdateWindow	Draws the client area
WinMain	Main procedure
<b>New structures</b>	<b>Brief description</b>
MSG	Data structure for messages
WNDCLASS	Data structure for the class

## Source code: FIRST.C

```
/** FIRST.C ****
/** Short Windows application. FIRST.C displays a window with title bar. */
/** The title bar states, "This is my first application" */
****

#include "windows.h"                                // Include windows.h header file

BOOL FirstInit ( HANDLE );
long FAR PASCAL FirstWndProc( HWND, unsigned, WORD, LONG);

/** WinMain (main function for every Windows application) ****
                                         // Blank line for readability
int PASCAL WinMain( hInstance, hPrevInstance, lpszCmdLine, cmdShow )
HANDLE hInstance, hPrevInstance;                  // Current & previous instances
LPSTR lpszCmdLine;                            // Long ptr to string after
                                              // program name during execution
int cmdShow;                                  // Specifies the application
                                              // window's appearance
{
    MSG msg;                                // Message variable
    HWND hWnd;                             // Window handle

    if (!hPrevInstance)                      // Initialize first instance
    {
        if (!FirstInit( hInstance )) // If initialization fails
            return FALSE;                // return FALSE
    }
/** Specify appearance of application's main window ****

    hWnd = CreateWindow("First",
                        "This is my first application",      // Window caption
                        WS_OVERLAPPEDWINDOW,                 // Overlapped window style
                        CW_USEDEFAULT,                     // Default upper-left x pos.
                        0,                               // Upper-left y pos.
                        CW_USEDEFAULT,                     // Default initial x size
```

```
        0,                                     // y size
        NULL,                                    // No parent window
        NULL,                                    // Window menu used
        hInstance,                                // Application instance
        NULL);                                   // No creation parameters

    ShowWindow( hWnd, cmdShow );           // Make window visible
    UpdateWindow( hWnd );                 // Update window

    while (GetMessage(&msg, NULL, 0, 0)) // Message reading
    {
        TranslateMessage(&msg);          // Message translation
        DispatchMessage(&msg);         // Send message to Windows
    }

    return (int)msg.wParam;                // Return wParam of last message
}

BOOL FirstInit( hInstance )           // Initialize instance handle
HANDLE hInstance;                   // Instance handle
{

/** Specify window class *****/
    WNDCLASS      wcFirstClass;           // Main window class

    wcFirstClass.hCursor     = LoadCursor( NULL, IDC_ARROW );
                               // Mouse cursor
    wcFirstClass.hIcon       = LoadIcon( NULL, IDI_APPLICATION );
                               // Default icon
    wcFirstClass.lpszMenuName = NULL;
                               // No menu
    wcFirstClass.lpszClassName = "First";
                               // Window class
    wcFirstClass.hbrBackground = GetStockObject( WHITE_BRUSH );
                               // White background
    wcFirstClass.hInstance   = hInstance;    // Instance
    wcFirstClass.style      = 0;           // Horizontal & vertical redraw
                                         // of client area
    wcFirstClass.lpfnWndProc = FirstWndProc; // Window function
    wcFirstClass.cbClsExtra  = 0;
                               // No extra bytes
    wcFirstClass.cbWndExtra  = 0;
                               // No extra bytes

    if (!RegisterClass( &wcFirstClass )) // Register window class
        return FALSE;                  // Return FALSE if registration fails
                                         // If registration is successful,
    return TRUE;                     // Return TRUE
}
```

```
/** FirstWndProc ****
** Main window function: All messages are sent to this window      */
/********************************************

long FAR PASCAL FirstWndProc( hWnd, message, wParam, lParam )
HWND    hWnd;                                // Window handle
unsigned message;                           // Message type
WORD     wParam;                            // Message-dependent 16 bit value
LONG    lParam;                            // Message-dependent 32 bit value
{

    switch (message)                      // Process messages
    {
        case WM_DESTROY:                // Send WM_QUIT if window is
            PostQuitMessage(0);        // destroyed
            break;                     // End of this message process

        default:                      // Send other messages to
            return (DefWindowProc( hWnd, message, wParam, lParam )); // default window function
            break;                     // End of this message process
    }
    return(0L);
}
```

## Module definition file: FIRST.DEF

NAME	First
DESCRIPTION	'1st Windows Application'
EXETYPE	WINDOWS
STUB	'WINSTUB.EXE'
CODE	PRELOAD MOVEABLE DISCARDABLE
DATA	PRELOAD MOVEABLE MULTIPLE
HEAPSIZE	4096
STACKSIZE	4096
EXPORTS	FirstWndProc @1

# Compiling and linking

You'll need the FIRST.C source code and FIRST.DEF module definition file (these are included on the companion diskette which accompanies this book). You must specify three options in order to compile the source text. The -c switch indicates that the source text should be compiled instead of linked.

Since all the structures used in Windows must be packed, the -Zp switch packs structures. The -Gw switch inserts a prolog or an epilog before and after each function that is defined with FAR.

This additional code ensures that the data segment and the stack are used correctly. This is important because a DLL cannot have its own stack. Instead, it uses the stack of the application.

The -Gs switch for disabling checks for stack overflow is sometimes combined with the switches previously described.

You would enter the following line from the DOS prompt to compile the program:

```
cl -c -Gw -Zp first.c
```

During linking you must specify the ALIGN:16 switch in addition to the application name. This switch indicates that the code and data segments must be aligned to 16 bytes. Also, you must include the import library for Windows (LIBW.LIB) and the name of the definition file.

```
link /align:16 first,first.exe,,libw+slibcew,first.def
```

One final, vital step is needed. The application as it stands will run, but gives you an Application Compatibility Warning dialog box. This means that the application was intended for an earlier version of Windows. The final step is to run the FIRST.EXE file through the resource compiler RC.EXE.

```
rc first.exe
```

You can now add the FIRST.EXE program to its own program group.

You can perform the same task from within the directory containing the files you wish to compile by creating a batch file. We created a file named COMPILE.BAT, which looks like this:

```
cl -c -Gw -Zp %1  
link /align:16 %1,%1.exe,,libw+slibcew,%1.def  
rc %1.exe
```

You can enter this file using the DOS COPY CON command or any text editor or word processor that generates ASCII files. Save this file to a directory contained in your path. Then when you are in the directory containing your source and definition files, type the following and press **[Enter]** to compile your program:

```
compile first
```

The batch file performs all the necessary tasks.

## How FIRST.EXE works

Now that you've seen the source codes for FIRST.C, let's look at the other elements needed to make this a Windows application.

## Header file

The header file WINDOWS.H, found in the Software Development Kit (SDK), must be included in every Windows application. The prototypes of all Windows functions, many data types, the messages, and other constants are defined in this header file.

If you examine this file, you'll get an overview of these definitions. With some practice, these new data types are often easier to read and understand than the standard types used in the C language. For example, the type LPSTR is simply a far pointer to a string and is defined in WINDOWS.H; with char far \*.

Many typedef definitions, which define different handles, exist in this header file. These handles are used to administer loaded objects. The Windows system stores the information about these objects in an internal table. Handles are references to entries in this table. Besides the general handle, the following handles also exist:

HBITMAP	Handle for a physical bitmap
HBRUSH	Handle for a physical brush
HCURSOR	Handle for a cursor resource
HDC	Handle for a display context
HFONT	Handle for a physical font
HICON	Handle for an icon resource
HMENU	Handle for a menu resource
HPALETTE	Handle for a logical palette
HPEN	Handle for a physical pen
HRGN	Handle for a physical region
HSTR	Handle for a string resource

All the handles are the same size, which is the size of the data type WORD. So that the program listings are easy to read, always use the right handle type.

## WinMain

Since the WinMain function is the point of entry for all applications, a Windows application must have this function. The WinMain function replaces the main() function found in standard C programs.

In most Windows applications WinMain initializes the window class, then creates and displays a window. A call is then made to the message loop. This call is only exited when the application is closed.

WinMain always has the same structure:

```
int PASCAL WinMain( hInstance, hPrevInstance, lpszCmdLine, CmdShow )
```

The words in this line mean the following:

PASCAL: This indicates that the parameters are stored from left to right on the stack and that the function that is called clears the stack.

In the C language the opposite is normally used. However, since Windows calls the WinMain function directly and always uses this convention, the keyword PASCAL is needed.

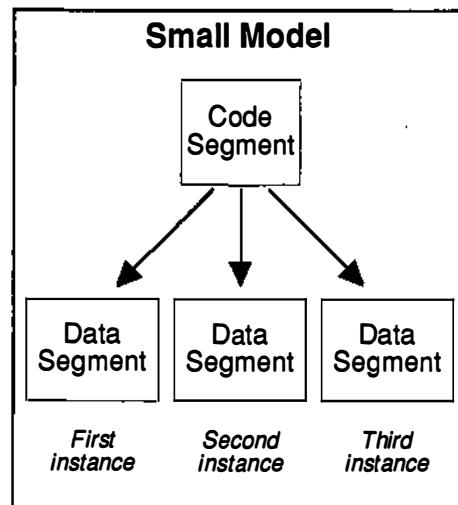
**hInstance, hPrevInstance:**

These variables clearly identify the application that was just started under Windows. Each time an application is started, this startup is called an instance. If the application is started for a second time, a new hInstance is assigned to the new copy. So each copy is a new instance.

This handle can be compared with a task ID in other multitasking systems. The variable hPrevInstance is a handle to the previous instance. If the application was started for the first time, the hPrevInstance always contains the value NULL. Windows administers these parameters.

It's also important that the application's code is stored only once in memory. This is true even if the application is started several times. Therefore, the code must be reentrant. All Windows functions have this characteristic.

Reentrant code saves storage space. Only the data segments are different for each instance of the application.



- lpszCmdLine: The third parameter is a far pointer, which points to a string that is terminated by a NULL. Here arguments can be passed when the program begins execution.
- nCmdShow: This integer value determines whether the application should be started as a window or as an icon, depending on the user's preference. If the window should be displayed, the value is passed to the ShowWindow function.

## Initialization

Before creating a window, you must register a class. For this, you need the structure WNDCLASS defined in the WINDOWS.H header file.

```
typedef struct tagWNDCLASS {
    LPSTR lpszClassName;           // Points to class name
    long (FAR PASCAL *lpfnWndProc)(); // Points to Window function
    HANDLE hinstance;              // Handle of Instance which
                                   // registers class
    int cbClsExtra;                // Additional bytes, assigned to
                                   // class structure
    int cbWndExtra,                 // Additional bytes, assigned to all
                                   // Windows structures relative to
                                   // this class
    HICON hIcon;                   // Handle to icon which represents
                                   // minimized window
    HCURSOR hCursor;               // Handle to cursor displayed in window
    HBRUSH hbrBackground;          // Handle to brush, which determines
                                   // window background color
    LPSTR lpszMenuName;            // Points to menu name when menu present
    Word style;                    // Style type
} WNDCLASS;
```

One window class describes the basic appearance of the window, which relates to this class and contains a pointer which points to the window function that belongs to it. Every class is assigned to one window function. Since there are some predefined classes, there are also some predefined window functions, such as the Button class.

In our first application, the WNDCLASS structure contained the following values:

lpszClassName:	"First"; the name is usually identical to the application name.
lpfnWndProc:	FirstWndProc; the name can be selected at random.
hInstance:	Handle to instance which registers class.
style:	NULL; no style is used.
hbrBackground:	GetStockObject(WHITE_BRUSH); this indicates a white background because GetStockObject returns a handle to the white standard brush.
hCursor:	LoadCursor(NULL, IDC_ARROW); the standard arrow, which is designated by the constant IDC_ARROW, is used as Cursor.
hIcon:	LoadIcon(NULL, IDI_APPLICATION); the constant IDI_APPLICATION designates a small square with a frame, which is displayed when a window is minimized.
lpszMenuName:	NULL; no menu.
cbClsExtra:	NULL; no extra bytes used.
cbWndExtra:	NULL; no extra bytes used.

The first three fields must contain the appropriate values, from which the names for the class and for the window function can be selected. Once the structure WNDCLASS has been provided, the actual registration can be performed with the RegisterClass function. A pointer to the structure is returned.

After a successful registration, a jump to the main program is performed with TRUE. If the registration fails, FALSE is used.

The return value should be queried with WinMain because it's useless to create a window if its class isn't available.

# Window creation

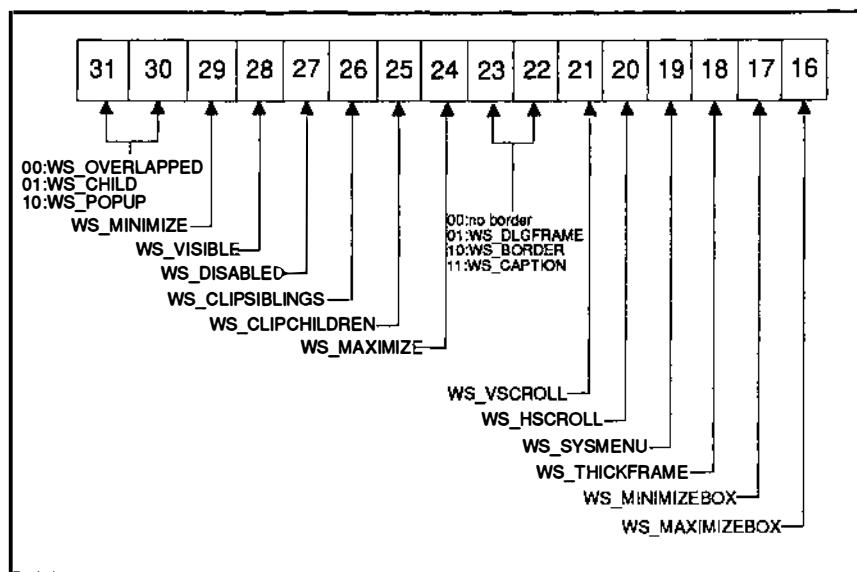
Let's take a look at how a window is created based on the initialization information we previously discussed. Here we've listed the standard syntax for window generation, along with the code used in the FIRST.C source code:

Syntax	Example	// Comments
CreateWindow(	CreateWindow(	// CreateWindow function
lpClassName,	"First",	// Specifies class name
lpWindowName,	"This is my first application",	// Title bar
dwStyle,	WS_OVERLAPPEDWINDOW,	// Window style
X, Y,	CW_USEDEFAULT, 0,	// Default window placing
nWidth, nHeight,	CW_USEDEFAULT, 0,	// x and y sizes
hWndParent,	NULL,	// No parent window
hMenu,	NULL,	// No menu
hInstance,	hInstance,	// Application instance
lpParam);	NULL);	// No creation parameters

lpClassName: This name must exactly match the one in the WNDCLASS structure, otherwise a window won't be created.

lpWindowName: Text that appears in the title bar.

dwStyle: Window Styles; this is a 32 bit field that Windows interprets as a number of switches. These flags are given names in WINDOWS.H and linked with a bitwise Or. Since the upper 16 bits are usually the more interesting flags, they are the only ones described in the illustration.



A style called WS\_OVERLAPPEDWINDOW appears in the listing. This style is a combination of the WS\_CAPTION, WS\_SYSMENU, WS\_MINIMIZEBOX, WS\_MAXIMIZEBOX, WS\_THICKFRAME, and WS\_OVERLAPPED styles. Based on these styles, the window has a title bar, a system menu, a minimize box, and a maximize box. This window is called a main window and can be resized.

The WS\_OVERLAPPED style is used to create a main window. These windows are usually opened once in WinMain and then remain open until the end of the application. Normally the user can change the size and position of a main window on the screen.

In addition to main windows, child windows and popup windows are also possible.

## Child windows

Child windows are always subordinate to main windows. These child windows are always found within the client area of the parent window. If the parent window is minimized, any part of the child window that overlaps the parent window will be truncated.

To create a child window you must use the WS\_CHILD style. Child windows are often used to divide the client area of the parent window into several areas.

## Popup windows

The WS\_POPUP style creates popup windows. These windows are divided into two groups: those with parents and those without parents. The first group (with parents) is also subordinate to a main window. However, unlike child windows, popup windows can have their own menus.

The second group (without parents) can be moved without being limited by the main window. Also, if the main window is closed, this type of popup window isn't affected. The only difference between an overlapped window and a popup window is that you cannot minimize a popup window. For example, all dialog boxes are popup windows.

X, Y, nWidth, nHeight:

The X and Y parameters specify the location of the upper-left corner of the window. For main windows and popup windows without parents, the reference point is the upper-left corner of the screen. For popup windows with parents and child windows, the coordinates are relative to the origin of the parent client area. The other parameters specify the size of the window. The CW\_USEDEFAULT value appears in the listing. This value indicates that Windows should place the window in a default position and use the default size. These default values are based on the number of applications that are already started.

**hWndParent:** Handle of the parent window; If you're creating a child window or popup window with parents, you must specify the handle of the parent window here to create the correct reference.

**hMenu:** Menu handle; If the window is supposed to contain a menu, you can write the handle here. This handle is created when the menu is loaded.

**hInstance:** Instance handle; You must specify to which instance the window belongs. This ensures that data from the proper data segment is used.

**lpParam:** Additional data that can be used by the window function when the window is created.

If CreateWindow is able to create a window, it returns a handle to this new window. Many subsequent functions will need this handle. If a window cannot be created, the return value of the function is NULL.

When CreateWindow creates a window, the window isn't automatically visible on the screen. First the window is displayed with the ShowWindow and UpdateWindow functions. Since Windows doesn't handle the workspace, ShowWindow is responsible for drawing the window without the client area.

The second parameter specifies whether the window appears as an icon or as a normal window. This information is located in the CmdShow variable, which is the fourth parameter of WinMain. Instead of CmdShow, constant values, such as SW\_SHOWMINNOACTIVE, are specified for any windows other than the main window. This constant indicates that the window should be displayed as an icon and that the previously active window should remain active.

UpdateWindow prompts the application (actually the main window function) to paint its own workspace (the client area). In the sample application, the window background is filled with white paint.

## MSG structure

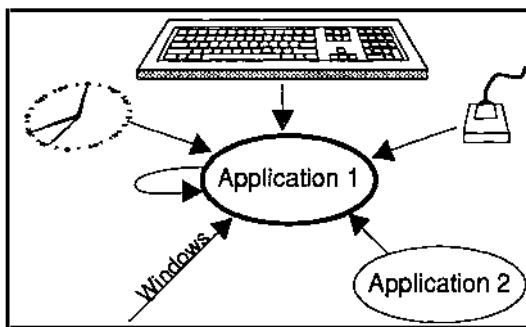
The MSG structure is very important to the rest of the process. This structure contains information, from the application message queue, that is processed by the window function.

## Message queues

Windows is a message oriented system. This means that all the information within the system is exchanged by sending and receiving messages.

For example, some type of input generates messages, which are temporarily stored by Windows. Then, when the appropriate window function has the opportunity, the messages are processed. In this way the application is separate from the input coming from the outside.

As you can see in the following illustration, messages can originate from the keyboard, mouse, timer, Windows system, another application, or the application itself.



Windows manages all of the messages by using message queues. There is a system message queue for the entire system and an application message queue for each application. Think of these queues as a circular buffer.

### System message queue:

All mouse, keyboard, and timer events are placed in this queue by the mouse, keyboard, or system driver with the help of functions from the USER.EXE library. Another part of the USER.EXE library takes the message from the buffer, translates it into the predefined MSG structure, and places it in the proper application message queue.

## Capture

Windows must pass all the mouse messages to the application that currently owns the mouse (i.e., the application that has the mouse *capture*). Usually a window owns the mouse capture as soon as the pointer is over the window. Windows places all the messages, which could result from the mouse, in the window's application message queue.

You can also retain the mouse capture by using the SetCapture function when the mouse leaves the window.

## Focus

Windows always knows which window currently owns the keyboard. This is also referred to as the *focus*. Usually the active window, which can be identified by the color of its title bar, has the focus.

You can pass the focus to a new window by using certain key combinations (**Alt** + **Esc** or **Alt** + **Tab**) or by clicking the new window with the mouse.

When you press a key, Windows first places this event in the system message queue and then passes it to the appropriate application message queue.

Frequently, the focus and mouse capture are assigned to different windows. However, only one window can have the focus or mouse capture at a given time.

Now let's return to the MSG structure. This structure consists of six fields, but the first four are the most important. The application message queue supplies values to the MSG structure.

```
typedef struct tagMSG {  
    HWND    hwnd;  
    WORD    message;  
    WORD    wParam;  
    LONG    lParam;  
    DWORD   time;  
    POINT   pt;  
} MSG;
```

**hwnd:** Identifies the window that receives the message. All instances of an application use only one application message queue. The window handle also identifies the instance since the relationship between the instance and window is established through the CreateWindow function.

message:	Specifies the message. This field contains a message number. In the WINDOWS.H header file, a name is included with the message number.
wParam, lParam:	Although these fields can contain additional information, whether they do depends on the message. For example, with keystrokes it's useful to know which key was pressed.
time:	Specifies the time the message was taken.
pt:	Specifies the position of the mouse pointer when the message was taken.

## Message loop

The message loop supplies the MSG structure. This loop is created by a while instruction and always contains at least two functions: GetMessage (or PeekMessage) and DispatchMessage.

The GetMessage function takes the next message in the application message queue and places it in the MSG structure. Therefore, the first parameter is also the address of the MSG variables. The other parameters are set to NULL, which indicates that all the messages should be processed.

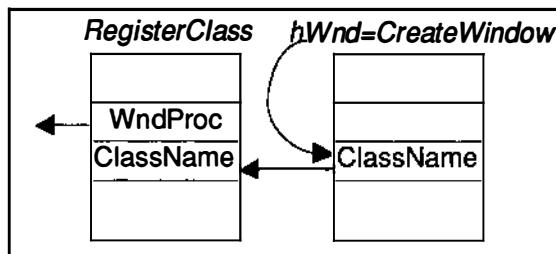
However, if there isn't a message in the application message queue, the application relinquishes control of the CPU to Windows. Now Windows can pass control of the CPU to the next application, which then processes its queue.

Meanwhile, the first application remains at the GetMessage function and waits for its next turn. Windows uses a chained task list to manage the applications that are waiting. This is referred to as non-preemptive multitasking.

Therefore, a running task that has the CPU and messages to be processed isn't interrupted after a specified time, as with preemptive time slice multitasking. Instead, the task is interrupted only when it has no more messages to process.

The actual processing of the messages occurs in the window function of the appropriate window instead of in WinMain. The message is passed to this window procedure with the help of the DispatchMessage function.

Using the internal structures that are created when you call the RegisterClass and CreateWindow functions, Windows knows which window function is the correct one. Windows is able to access the class name and find the correct window procedure from the hWnd handle, which is the first parameter of the MSG variable.



The message loop may contain other functions besides GetMessage and DispatchMessage. Often the TranslateMessage function is located between these two functions. This function is needed to process keyboard input. The chapter describing input provides additional information on this subject.

The message loop ends when GetMessage FALSE is the return value. For example, this occurs when the user closes the application. The WM\_QUIT message is then sent. We'll explain the exact sequence later.

## Window functions

Each Windows application must have at least one window function. However, the size of this function can vary greatly.

```
long FAR PASCAL FirstWndProc( hWnd, message, wParam, lParam )
```

The transfer parameters of each window function are identical to the first four parameters of the MSG structure because the messages and additional information are processed here in the window procedure. The window procedure is loaded to main memory only once and is used by all the instances of an application.

Windows is able to determine the hInstance from the internal structure via the hWnd parameter. This parameter enables Windows to differentiate between individual instances.

All window functions must be declared as FAR PASCAL. As described earlier under WinMain, PASCAL declares that the parameters be placed on the stack from left to right and that the called function clean up the stack. FAR is needed because an intersegment jump (a jump to another segment) occurs when Windows calls the window procedure.

A switch instruction branches to the appropriate messages in the window procedure. When this happens, only the messages that are important to the application must be processed. For example, often all the messages from the mouse are unimportant. All unnecessary messages are simply passed to a default window procedure. Usually the WM\_DESTROY message results in the following reaction:

```
case WM_DESTROY:  
PostQuitMessage(0);  
break;
```

## Default window functions

```
LONG DefWindowProc( hWnd, wMsg, wParam, lParam )
```

This function is already defined in Windows and is called DefWindowProc. It performs predefined actions for certain messages that must be answered.

For example, pressing the left mouse button usually activates the application over whose window the mouse is currently located. Actually, pressing the left mouse button results in the WM\_LBUTTONDOWN message, which is then processed by the default window procedure. The default window function also ignores many other messages.

## More about messages

In Windows all the messages are divided into eleven groups. These groups provide additional information about the messages:

- Window Management Messages
- Initialization Messages
- Input Messages
- System Messages
- Clipboard Messages
- System Information Messages
- Control Messages
- Notification Messages
- Scrollbar Messages
- Non-Client Area Messages
- Multiple Document Interface Messages

All the messages we've discussed so far begin with the letters "WM", which is an abbreviation for "Window Message". These messages apply to any window. However, there are other messages that are only relevant to special windows. For example, the message BM\_SETCHECK represents a button that has been clicked.

All of these messages are system messages with a value between 0 and WM\_USER-1. Similar to all the other messages, WM\_USER is also defined in the WINDOWS.H header file. It's also possible to create your own message system. The hexadecimal values for your self-defined messages must be between WM\_USER and 7FFFH.

There are also queued and nonqueued messages. A queued message is located in the application message queue. To obtain a queued message, use GetMessage. Use DispatchMessage to pass a queued message to the window procedure. Instead of being in the queue, a nonqueued message is passed directly to the window function. Usually messages that should be processed quickly are nonqueued.

For example, the WM\_CREATE message is a nonqueued message. This message is a result of the CreateWindow function reference and is the first message to be processed in the window procedure. You can also send both queued and nonqueued messages to an application.

The function PostMessage and SendMessage are responsible for sending messages to an application. As their names indicate, PostMessage places the message in the queue and SendMessage sends the message to the window. Both functions have the same parameters as a window function:

SendMessage( hWnd, message, wParam, lParam ) - Window function

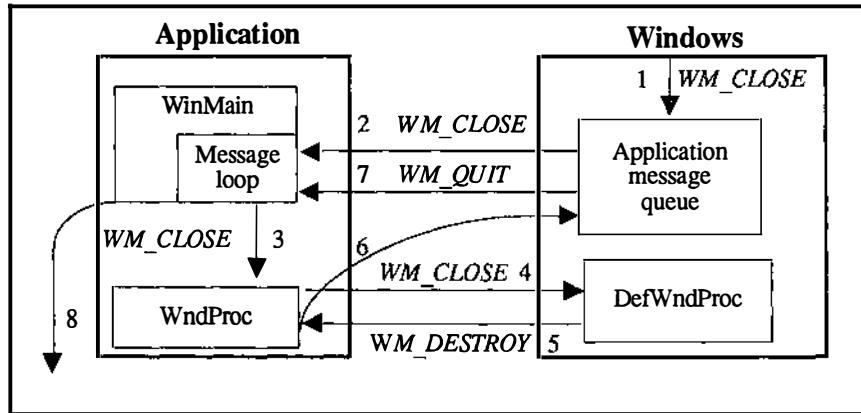
or

PostMessage( hWnd, message, wParam, lParam ) - Application message queue

## Exiting an application

In this section we'll use an example to demonstrate how messages are processed and linked.

As you can see in the following illustration, when the user exits the application (e.g., by pressing **Alt** + **F4**), Windows generates the WM\_CLOSE message. Then after a series of other messages, Windows places the WM\_CLOSE message in the application message queue.



The `GetMessage` function takes `WM_CLOSE` from the queue and `DispatchMessage` passes it to the window procedure.

In smaller applications, instead of being processed, `WM_CLOSE` is passed to the default window procedure.

The default window procedure then calls the `DestroyWindow` function. This function destroys the window and sends the `WM_DESTROY` message directly to the window procedure without using the queue. Usually applications react to `WM_DESTROY` with the `PostQuitMessage` function.

This produces the WM\_QUIT message, which is placed in the application message queue. If GetMessage takes this message out of the queue, the return value of GetMessage is FALSE. When this happens, the message loop ends and the entire application is closed.

## The module definition file

In addition to the source text, the linker also needs a module definition file. The linker needs this file to link the compiled source text with Windows and other libraries so that an executable Windows application can be created.

Let's look again at the module definition file for our first application:

NAME	First
DESCRIPTION	'1st Windows Application'
EXETYPE	WINDOWS
STUB	'WINSTUB.EXE'
CODE	PRELOAD MOVEABLE DISCARDABLE
DATA	PRELOAD MOVEABLE MULTIPLE
HEAPSIZE	4096
STACKSIZE	4096
EXPORTS	FirstWndProc @1

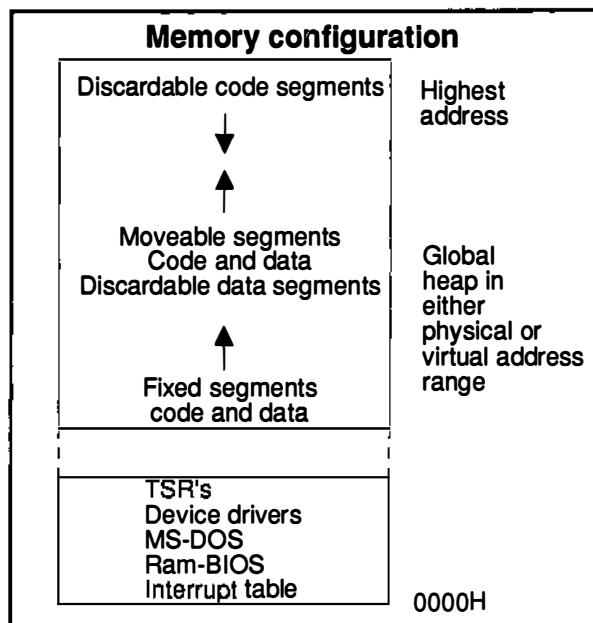
## Module definition file keywords

All module definition files contain specific keywords, which define information needed in compiling and linking the application. Here's a complete set of keywords:

<b>Keyword</b>	<b>Explanation</b>
NAME/LIBRARY	Module or DLL name
DESCRIPTION	One line description
EXETYPE	Windows or OS/2
STUB	"This program requires Microsoft Windows."
DATA	Data segment attribute
CODE	Code segment attribute
HEAPSIZE	Size of the local heap
STACKSIZE	Size of the local stack
SEGMENT	Additional code segment
EXPORTS	Export functions
IMPORTS	Import functions

Let's take a closer look at these keywords, in the order in which they appear in the file:

- NAME:** The name should always be the name of the Windows application; otherwise the linker emits a beep.
- LIBRARY:** To create a DLL, instead of the keyword NAME, use LIBRARY (more on this later).
- STUB:** The WINSTUB.EXE application already exists and is called when you try to start the Windows application from DOS. WINSTUB.EXE then displays the sentence "This program requires Microsoft Windows." After WINSTUB.EXE executes, the system returns to the DOS system prompt.
- CODE, DATA:** Code and data segments are both set to PRELOAD and MOVEABLE. Since the application was created using the SMALL model, it has exactly one code segment and one data segment. If the application doesn't currently have the CPU and Windows needs memory space for other applications, Windows can move both of these segments in memory. The CODE keyword can also include the DISCARDABLE parameter, which specifies that the code segment can be removed from memory when there is a shortage of memory space. To see how Windows divides the memory, refer to the following illustration:



If the application allocates a fixed segment along with all other moveable segments above the fixed segment, Windows attempts to move this fixed segment as close to the bottom of the heap as possible. This movement occurs while Windows rearranges all moveable or discardable segments.

Although rearranging the memory helps reduce fragmentation, the process is time-consuming. So, when you're programming you shouldn't use too many segments with the **FIXED** attribute.

The **MULTIPLE** option, under **DATA**, specifies that each time the application is restarted, creating a new instance, a new data segment is also created. Always specify this option for a Windows application. Since a DLL can only have one data segment, use **SINGLE** instead of **MULTIPLE**.

#### STACKSIZE:

Each Windows application must have a stack, whose size is specified in bytes. The stack is used, for example, as temporary storage for function arguments. At least 4096 bytes should be used for a

small Windows application. A DLL doesn't have its own stack.

**HEAPSIZE:** This is where the local heap is specified; 256 bytes is the minimum.

**EXPORTS:** This keyword defines the name and gives you the option of defining the ordinal numbers of all the functions to be exported. A function must be exported when Windows or another application calls it. In the first sample application it is only the FirstWndProc window procedure.

All the functions that Windows calls are referred to as callback functions. The ordinal number, which is another way to identify these functions, can accept any integer value.

**IMPORTS:** Here you must specify all functions called by the application that aren't defined in their source file or in the library statically linked to the application. These functions are defined by Dynamic Link Libraries (DLLs).

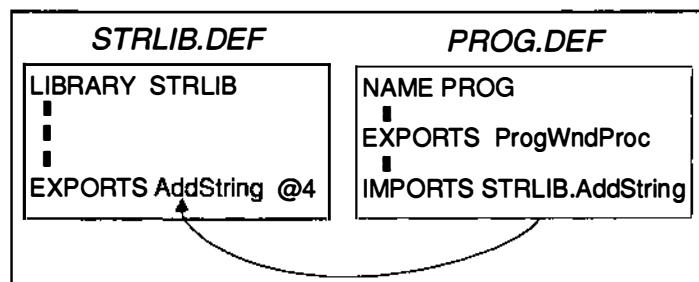
## **Dynamic Link Library (DLL)**

Windows contains the following Dynamic Link Libraries (DLLs): KERNEL.EXE, USER.EXE, and GDI.EXE. Unlike static libraries, DLLs aren't physically linked to the application. The application itself only contains cross references to the functions defined in a DLL. The appropriate DLL is loaded only when the application starts or when it's needed. Since the code of these functions must be reentrant, a DLL is loaded into memory only once, even if different applications access the same function. This saves memory space in RAM.

## **Example of IMPORTS and EXPORTS**

We'll use an example to demonstrate the relationship between the keywords IMPORTS and EXPORTS. First assume that a Windows application (PROG.EXE) is using a function, called AddString, from a

custom DLL (**STRLIB.EXE**). This makes the definition files of the DLL and the Windows application important.

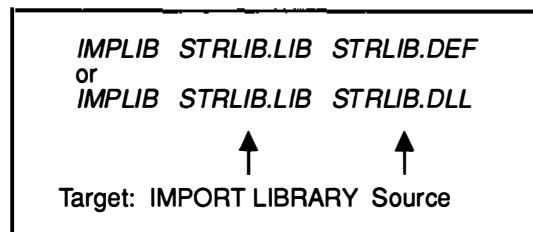


You can also use the ordinal number of the `AddString` function (e.g., `IMPORTS STRLIB.4`) with the `IMPORTS` keyword.

However, if the DLL consists of ten or twenty functions instead of a single function, and if the Windows application wanted to use all of these functions, a lot of writing would be required for the `IMPORTS` keyword.

Fortunately, you can use the `IMPORT LIBRARY` to solve this problem. Instead of code, an import library contains information about the desired function's DLL and ordinal number.

The Microsoft import library `LIBW.LIB` contains three Windows DLLs: `GDI.EXE`, `USER.EXE`, and `KERNEL.EXE`. Use the `IMPLIB.EXE` utility to create custom Dynamic Link Libraries.



When linking the Windows application `PROG.EXE`, you must specify the custom import library `STRLIB.LIB`, and `LIBW.LIB`. However, you can omit the `IMPORTS` keyword in the definition file `PROG.DEF`.

## Example of dynamic linking

The following example demonstrates how a function from a DLL is found while a Windows application is running.

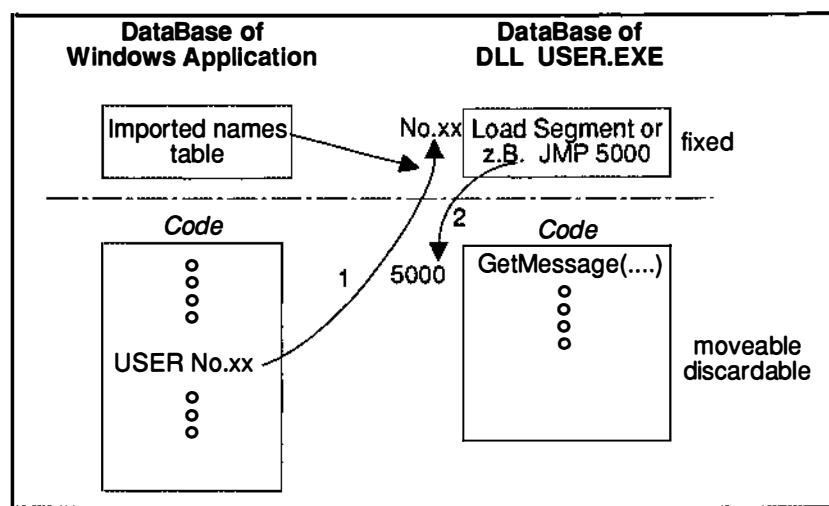
Each executable application has a header. Compared to a normal DOS program, a Windows application has additional information in this header. For example, all the functions to be imported are written in the imported names table of the linker. There are also other tables, such as the entry table and resource table.

The first time you start the Windows application this information is placed in a fixed memory segment called DataBase. Since the same thing occurs with a DLL, this also applies to DLL USER.EXE.

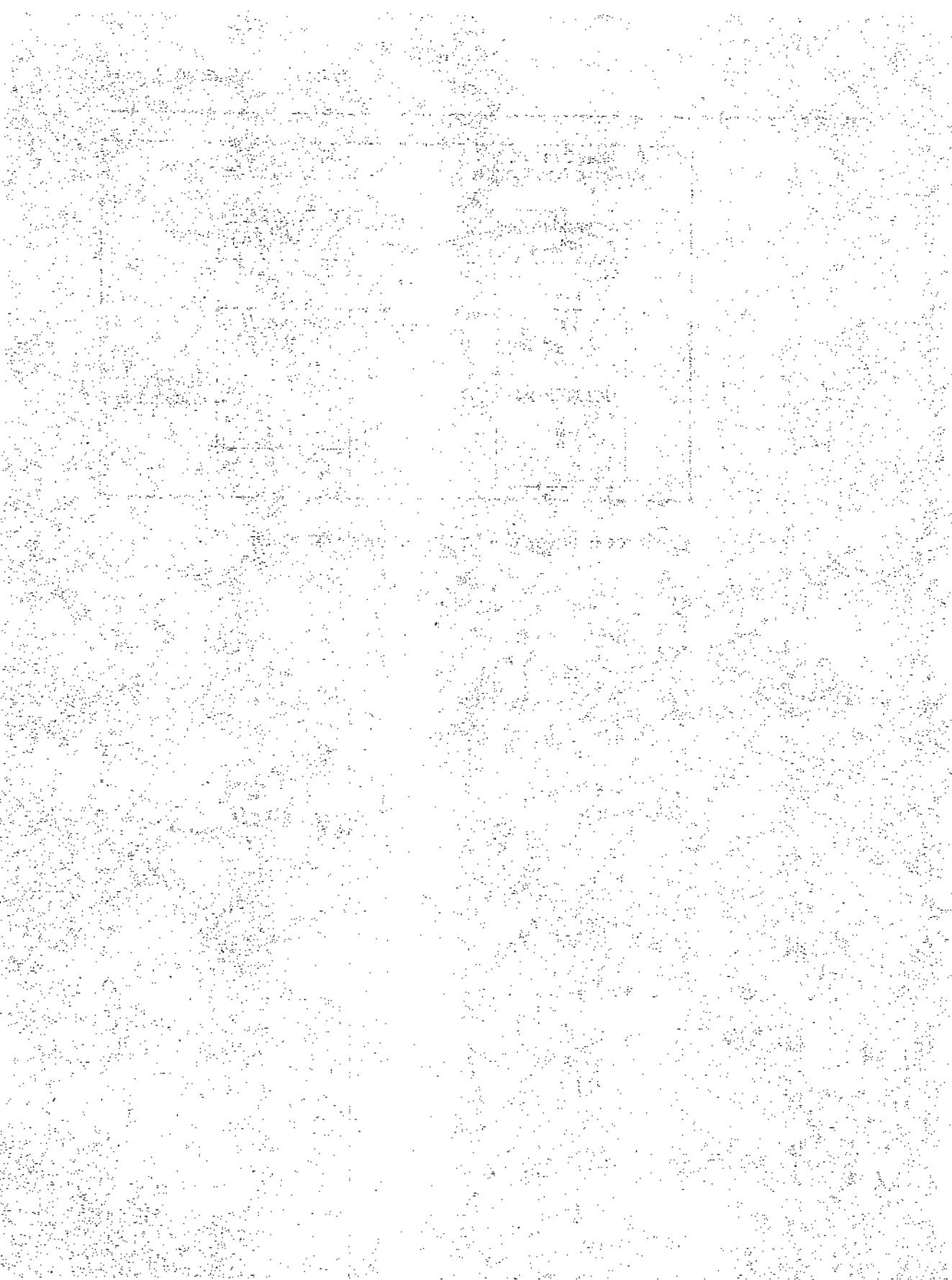
When you start the FIRST application, it will quickly call the GetMessage function. Since you're linking, you'll see the name and ordinal number of the DLL instead of the function reference. Windows can branch to the database of USER.EXE from the database of the Windows application.

There is a type of springboard for all the functions defined in this DLL. You can use either a jump to the address of the segment containing the definition of GetMessage or, if the necessary segment isn't currently in memory, the assembler command Load Segment.

If you use the second method, Windows loads this segment and changes the contents of the cell in the springboard to a jump to the currently loaded segment. This enables you to branch to the GetMessage function.



You'll see how you can write your own DLLs later.



# Keyboard and Mouse Input

## Input messages

In this chapter we'll describe the messages that can appear during input. Then we'll examine what happens when you press a key or use the mouse. Also, we'll see how to access the timer.

Whenever a user presses a key, moves the mouse, or presses a mouse button, Windows sends input messages to the application. Also, Windows generates a message in response to timer input. There are various kinds of messages:

Type of message	Description
Keyboard	User input from the keyboard
Character	Keyboard input translated into character code
Mouse	User input from the mouse
Menu	User input from a menu
Scroll bar	User input from a scroll bar
Timer	Input from the system timer

The keyboard, mouse, and timer messages are caused by a hardware interrupt. Windows places these messages in the system message queue, from which they are transmitted to the appropriate application message queue.

The character, menu, and scroll bar messages are a response to mouse or keyboard actions that occur in the non-client area or are caused by translated keyboard messages. Windows often sends these messages directly to the appropriate window function. We'll discuss the menu and scroll bar messages in later chapters.

## Keyboard input

Under Windows, applications aren't responsible for gathering input from the keyboard. Instead, the Windows system handles this task. The system then uses different messages to inform the application of the user's actions.

## Keyboard messages

Once a user presses or releases a key, Windows produces a keyboard message based on information from the keyboard driver. This message ultimately reaches the application that currently has the focus. The SetFocus function can assign the focus to a particular window. The GetFocus function retrieves the handle of the window, which currently owns the keyboard.

Message	Event
WM_KEYDOWN	Pressing a key
WM_KEYUP	Releasing a key
WM_SYSKEYDOWN	Pressing a system key
WM_SYSKEYUP	Releasing a system key

The WM\_SYSKEYDOWN and WM\_SYSKEYUP messages are usually more important to Windows than to the particular window function. They are caused by combinations of the **Alt** key and another key (e.g., **Alt** + **Tab** or **Alt** + **F4**). In certain instances these messages are processed.

However, if they are processed, they must be passed to the default window procedure. Otherwise, for example, you wouldn't be able to switch to another application when you press **Alt** + **Tab**. This would make basic operations impossible.

The other two messages appear when the **Alt** key isn't pressed. If, instead of being processed in the window function, they are passed to DefWndProc, they don't have any effect because Windows doesn't process them.

The virtual key code of the key that is pressed or released is located in the wParam parameter of every keyboard message. A virtual key code is a device independent value for a certain key.

The keys that, along with the letters and numbers, are frequently used have names defined in the WINDOWS.H header file. All the names begin with VK, VK\_F2, or VK\_INSERT.

The Param parameter is divided into five areas: number of repetitions, OEM scan code, context code, earlier key status, and transition code. However, often these values aren't used.

These four keyboard messages don't indicate whether or not the **Shift** key is also being pressed. However, there is a function that you can use to determine the state of every virtual key:

```
int GetKeyState( nVirtKey )
```

nVirtKey: Is the virtual key code (e.g., 'A' or VK\_SHIFT).

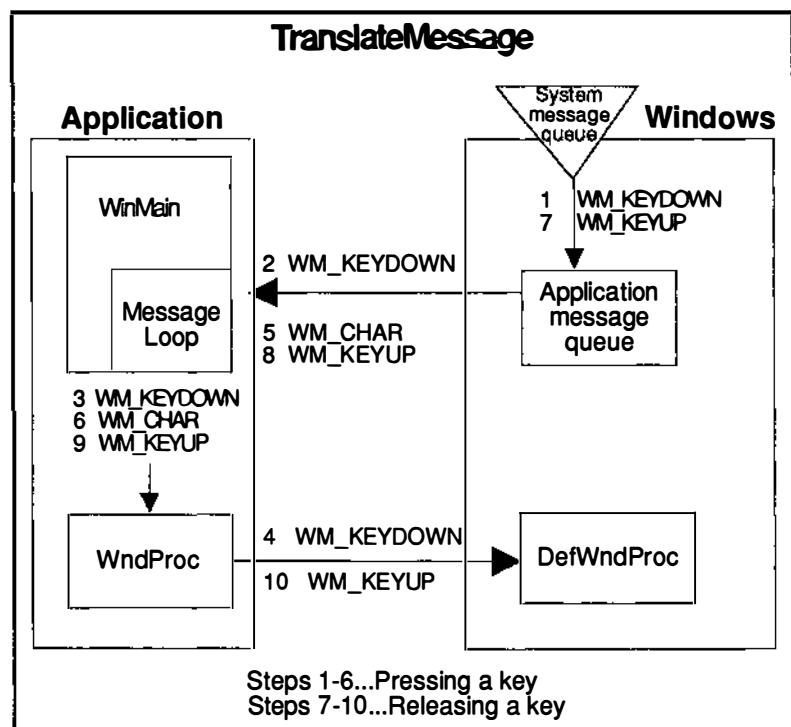
int: If a negative value is retrieved (i.e., if the highest bit is set to 1), then the specified key is pressed. Because of synchronization, you should use this function only in connection with the keyboard messages.

## Character messages

Many applications that work with the keyboard process character messages instead of keyboard messages. The TranslateMessage function is needed for this conversion and must be installed in the message loop. This function must create a new message, called WM\_CHAR or WM\_SYSCHAR, whenever it detects a WM\_KEYDOWN or WM\_SYSKEYDOWN message.

This new message is placed in the application message queue and is sent to the window procedure with the help of GetMessage and DispatchMessage.

The wParam parameter is very helpful because it contains the ANSI character code which can be prompted using a switch instruction.



The next example shows the process of pressing the **Shift** key and the **I** key to create an uppercase letter. However, instead of a WM\_CHAR message, both a WM\_KEYDOWN and a WM\_KEYUP message are produced for the **Shift** key.

Message	wParam
WM_KEYDOWN	virtual key VK_SHIFT
WM_KEYDOWN	virtual key "I"
WM_CHAR	ANSI Code "I"
WM_KEYUP	virtual key "I"
WM_KEYUP	virtual key VK_SHIFT

Windows applications normally use an extended character set called the ANSI character set. This character set matches the ASCII character set between the hexadecimal values of 20H and 7EH.

## ANSI Table

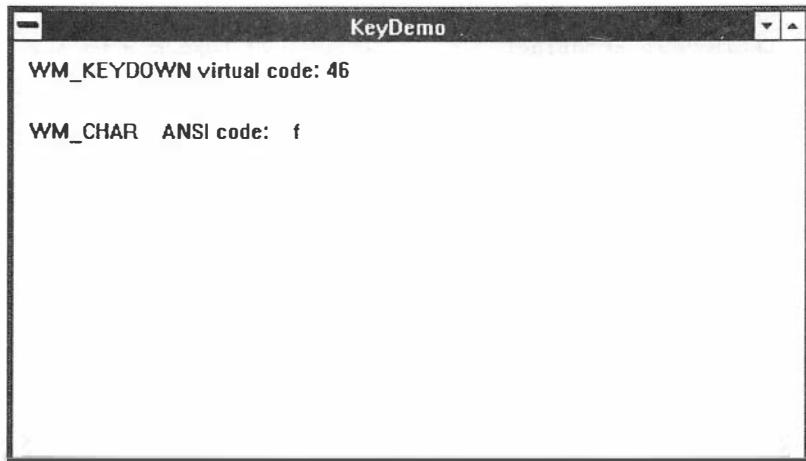
Dec.	Hex.	Character	Dec.	Hex.	Character	Dec.	Hex.	Character	Dec.	Hex.	Character
0 00	00	█	32 20	20	!	64 40	40	@	96 60	60	„
1 01	01	█	33 21	21	"	65 41	41	A	97 61	61	a
2 02	02	█	34 22	22	#	66 42	42	B	98 62	62	b
3 03	03	█	35 23	23	\$	67 43	43	C	99 63	63	c
4 04	04	█	36 24	24	%	68 44	44	D	100 64	64	d
5 05	05	█	37 25	25	&	69 45	45	E	101 65	65	e
6 06	06	█	38 26	26	'	70 46	46	F	102 66	66	f
7 07	07	█	39 27	27	(	71 47	47	G	103 67	67	g
8 08	08	█	40 28	28	)	72 48	48	H	104 68	68	h
9 09	09	█	41 29	29	,	73 49	49	I	105 69	69	i
10 0A	0A	█	42 2A	2A	*	74 4A	4A	J	106 6A	6A	j
11 0B	0B	█	43 2B	2B	+	75 4B	4B	K	107 6B	6B	k
12 0C	0C	█	44 2C	2C	,	76 4C	4C	L	108 6C	6C	l
13 0D	0D	█	45 2D	2D	-	77 4D	4D	M	109 6D	6D	m
14 0E	0E	█	46 2E	2E	.	78 4E	4E	N	110 6E	6E	n
15 0F	0F	█	47 2F	2F	/	79 4F	4F	O	111 6F	6F	o
16 10	10	█	48 30	30	0	80 50	50	P	112 70	70	p
17 11	11	█	49 31	31	1	81 51	51	Q	113 71	71	q
18 12	12	█	50 32	32	2	82 52	52	R	114 72	72	r
19 13	13	█	51 33	33	3	83 53	53	S	115 73	73	s
20 14	14	█	52 34	34	4	84 54	54	T	116 74	74	t
21 15	15	█	53 35	35	5	85 55	55	U	117 75	75	u
22 16	16	█	54 36	36	6	86 56	56	V	118 76	76	v
23 17	17	█	55 37	37	7	87 57	57	W	119 77	77	w
24 18	18	█	56 38	38	8	88 58	58	X	120 78	78	x
25 19	19	█	57 39	39	9	89 59	59	Y	121 79	79	y
26 1A	1A	█	58 3A	3A	:	90 5A	5A	Z	122 7A	7A	z
27 1B	1B	█	59 3B	3B	;	91 5B	5B	[	123 7B	7B	{
28 1C	1C	█	60 3C	3C	<	92 5C	5C	\	124 7C	7C	
29 1D	1D	█	61 3D	3D	=	93 5D	5D	]	125 7D	7D	}
30 1E	1E	█	62 3E	3E	>	94 5E	5E	^	126 7E	7E	~
31 1F	1F	█	63 3F	3F	?	95 5F	5F	_	127 7F	7F	█

**ANSI Table (continued)**

Dec.	Hex.	Character									
128	80	█	160	A0	:	192	C0	À	224	E0	à
129	81	█	161	A1	;	193	C1	Á	225	E1	á
130	82	█	162	A2	¢	194	C2	Â	226	E2	â
131	83	█	163	A3	£	195	C3	Ã	227	E3	ã
132	84	█	164	A4	¤	196	C4	Ä	228	E4	ä
133	85	█	165	A5	¥	197	C5	Å	229	E5	å
134	86	█	166	A6	:	198	C6	È	230	E6	ë
135	87	█	167	A7	\$	199	C7	Ҫ	231	E7	ç
136	88	█	168	A8	"	200	C8	ܶ	232	E8	ܵ
137	89	█	169	A9	ܹ	201	C9	ܸ	233	E9	ܵ
138	8A	█	170	AA	ܷ	202	CA	ܹ	234	EA	ܵ
139	8B	█	171	AB	ܸ	203	CB	ܹ	235	EB	ܵ
140	8C	█	172	AC	ܻ	204	CC	ܹ	236	EC	ܵ
141	8D	█	173	AD	-	205	CD	ܹ	237	ED	ܵ
142	8E	█	174	AE	ܹ	206	CE	ܹ	238	EE	ܵ
143	8F	█	175	AF	-	207	CF	ܹ	239	EF	ܵ
144	90	█	176	B0	ܷ	208	D0	ܹ	240	F0	ܵ
145	91	'	177	B1	ܹ	209	D1	ܹ	241	F1	ܵ
146	92	'	178	B2	ܷ	210	D2	ܹ	242	F2	ܵ
147	93	"	179	B3	ܹ	211	D3	ܹ	243	F3	ܵ
148	94	"	180	B4	ܻ	212	D4	ܹ	244	F4	ܵ
149	95	•	181	B5	ܹ	213	D5	ܹ	245	F5	ܵ
150	96	-	182	B6	ܹ	214	D6	ܹ	246	F6	ܵ
151	97	-	183	B7	ܷ	215	D7	ܹ	247	F7	ܵ
152	98	█	184	B8	ܷ	216	D8	ܹ	248	F8	ܵ
153	99	█	185	B9	ܹ	217	D9	ܹ	249	F9	ܵ
154	9A	█	186	BA	ܹ	218	DA	ܹ	250	FA	ܵ
155	9B	█	187	BB	ܹ	219	DB	ܹ	251	FB	ܵ
156	9C	█	188	BC	ܹ	220	DC	ܹ	252	FC	ܵ
157	9D	█	189	BD	ܹ	221	DD	ܹ	253	FD	ܵ
158	9E	█	190	BE	ܹ	222	DE	ܹ	254	FE	ܵ
159	9F	█	191	BF	ܹ	223	DF	ܹ	255	FF	ܵ

The values from 0 to 1FH (31 decimal) are undefined characters that are used differently by different devices.

## Keyboard example



The KEYDEMO.EXE application responds to each keypress, each release of a key, and to the WM\_CHAR message. In the client area, the screen displays the appropriate message and the value located in Param.

### New messages

WM\_KEYDOWN  
WM\_KEYUP  
WM\_CHAR

### Brief description

Sent when a key is pressed  
Sent when a key is released  
Results from the translation of WM\_KEYDOWN

### New functions

wsprintf

### Brief description

Formats and saves characters and values in a buffer

## Source code: KEYDEMO.C

```
/** KEYDEMO.C ****
** Displays WM_KEYDOWN and WM_KEYUP messages as virtual and ANSI codes   */
/** ****

#include "windows.h"                                // Include WINDOWS.H header file
#include "string.h"                                 // Include STRING.H header file
```

## Keyboard and Mouse Input

---

```
BOOL KeyInit ( HANDLE );                                // Key initialization
long FAR PASCAL KeyWndProc( HWND, unsigned, WORD, LONG);

/** WinMain (main function for every Windows application) *****/
int PASCAL WinMain( hInstance, hPrevInstance, lpszCmdLine, wCmdShow )
    HANDLE  hInstance, hPrevInstance;      // Current & previous instances
    LPSTR   lpszCmdLine;                 // Long ptr to string after
                                         // program name during execution
    int      wCmdShow;                  // Specifies the application
                                         // window's appearance
{
    MSG msg;                           // Message variable
    HWND hWnd;                         // Window handle

    if (!hPrevInstance)                // Initialize first instance
    {
        if (!KeyInit( hInstance ))     // If initialization fails
            return FALSE;             // return FALSE
    }
}

/** Specify appearance of application's main window *****/
hWnd = CreateWindow("KeyDemo",                      // Window class name
                    "KeyDemo",                   // Window caption
                    WS_OVERLAPPEDWINDOW,        // Overlapped window style
                    CW_USEDEFAULT,              // Default upper-left x pos.
                    0,                          // Upper-left y pos.
                    CW_USEDEFAULT,              // Default initial x size
                    0,                          // y size
                    NULL,                      // No parent window
                    NULL,                      // Window menu used
                    hInstance,                 // Application instance
                    NULL);                     // No creation parameters

ShowWindow( hWnd, wCmdShow );                      // Make window visible
UpdateWindow( hWnd );                            // Update window

while (GetMessage(&msg, NULL, 0, 0)) // Message reading
{
    TranslateMessage(&msg);           // Message translation
    DispatchMessage(&msg);          // Send message to Windows
}
return (int)msg.wParam;                           // Return wParam of last message
}

BOOL KeyInit( hInstance )                         // Key status
HANDLE hInstance;                               // Instance handle
```

```
{  
    /** Specify window class *****/  
  
    WNDCLASS      wcKeyClass;                      // Main window class  
  
    wcKeyClass.hCursor     = LoadCursor( NULL, IDC_ARROW );  
                           // Mouse cursor  
    wcKeyClass.hIcon      = LoadIcon( NULL, IDI_APPLICATION );  
                           // Default icon  
    wcKeyClass.lpszMenuName = NULL;                  // No menu  
    wcKeyClass.lpszClassName = "KeyDemo";            // Window class  
    wcKeyClass.hbrBackground = GetStockObject( WHITE_BRUSH );  
                           // White background  
    wcKeyClass.hInstance   = hInstance;                // Instance  
    wcKeyClass.style       = CS_VREDRAW | CS_HREDRAW;  
                           // Horizontal and vertical redraw of client area  
    wcKeyClass.lpfnWndProc = KeyWndProc;              // Window function  
    wcKeyClass.cbClsExtra  = 0;                      // No extra bytes  
    wcKeyClass.cbWndExtra  = 0;                      // No extra bytes  
  
    if (!RegisterClass( &wcKeyClass ) )               // Register window class  
        return FALSE;                                // Return FALSE if registration fails  
                                               // If registration is successful,  
    return TRUE;                                   // Return TRUE  
}  
  
/** KeyWndProc *****/  
/** Main window function: All messages are sent to this window */  
/** *****/  
  
long FAR PASCAL KeyWndProc( hWnd, message, wParam, lParam )  
{  
    HWND      hWnd;                      // Window handle  
    unsigned  message;                 // Message type  
    WORD     wParam;                   // Message-dependent 16 bit value  
    LONG     lParam;                   // Message-dependent 32 bit value  
  
    static char   chTextDown[35];        // Declare TextDown 35 characters  
    static char   chTextUp[35];         // Declare TextUp 35 characters  
    static char   chTextChar[35];        // Declare TextChar 35 characters  
    static BOOL    bDown = FALSE;        // Set Down default to FALSE  
    static BOOL    bUp   = FALSE;        // Set Up default to FALSE  
    static BOOL    bChar = FALSE;        // Set Char default to FALSE  
  
    HDC      hdc;                     // Device context handle  
    PAINTSTRUCT ps;                  // PAINTSTRUCT data structure  
  
    /** React to keyboard status *****/
```

## Keyboard and Mouse Input

---

```
switch (message)                                // Process messages
{
    case WM_KEYDOWN:                         // Process if key pressed
        wsprintf(chTextDown, "WM_KEYDOWN virtual code: %x", wParam);
        InvalidateRect(hWnd, NULL, TRUE); // Redraw client area
        bDown = TRUE;                      // Key pressed? Make bDown TRUE
        break;                            // End of this message process

    case WM_KEYUP:                           // Process if key released
        wsprintf(chTextUp, "WM_KEYUP   virtual code: %x", wParam);
        InvalidateRect(hWnd, NULL, TRUE); // Redraw client area
        bUp = TRUE;                      // Key released? Make bUp TRUE
        break;                            // End of this message process

    case WM_CHAR:                            // Process if key ANSI character
        wsprintf(chTextChar, "WM_CHAR     ANSI code:    %c", wParam);
        InvalidateRect(hWnd, NULL, TRUE); // Redraw client area
        bChar = TRUE;                     // Character key? Make bChar TRUE
        break;                            // End of this message process

    case WM_PAINT:                          // Redraw client area window
        hDC = BeginPaint (hWnd, &ps); // Begin WM_PAINT process
        if (bDown)                      // Key pressed?
        {
            // Display text...
            TextOut(hDC, 10, 10, chTextDown,
strlen(chTextDown));
            bDown = FALSE;             //... and reset bDown
        }
        if (bUp)                        // Key released?
        {
            // Display text...
            TextOut(hDC, 10, 30, chTextUp, strlen(chTextUp));
            bUp = FALSE;              //... and reset bUp
        }

        if (bChar)                      // Was key a character?
        {
            // Display text...
            TextOut(hDC, 10, 50, chTextChar,
strlen(chTextChar));
            bChar = FALSE;            //... and reset bChar
        }

        EndPaint(hWnd, &ps); // Finish processing WM_PAINT
        break;                      // End of this message process

    case WM_DESTROY:                       // Send WM_QUIT if window is
        PostQuitMessage(0); // destroyed
        break;                      // End of this message process
}
```

```

        default:          // Send other messages to
                           // default window function
        return (DefWindowProc( hWnd, message, wParam, lParam ));
        break;           // End of this message process
    }
    return(0L);
}

```

## Module definition file: KEYDEMO.DEF

NAME	KeyDemo
DESCRIPTION	'Keyboard example'
EXETYPE	WINDOWS
STUB	'WINSTUB.EXE'
CODE	PRELOAD MOVEABLE
DATA	PRELOAD MOVEABLE MULTIPLE
HEAPSIZE	4096
STACKSIZE	4096
EXPORTS	KeyWndProc @1

To compile and link this application, use the COMPILE.BAT batch file described earlier in this book. You can create this file using the DOS COPY CON command or text editor or word processor that generates ASCII files. Here's the listing again:

```

cl -c -Gw -Zp %1
link /align:16 %1,%1.exe,,libw+slibcew,%1.def
rc %1.exe

```

Save this file to a directory contained in your path. Then when you are in the directory containing your source and definition files, type the following and press **Enter**:

```
compile keydemo
```

The batch file performs all the necessary tasks.

## How KEYDEMO.EXE works

Four new messages have been added; these messages are processed in the application. We won't discuss the WM\_PAINT message in detail because this message is covered in the next chapter. For now, all you need to understand is that this is where the prepared text is displayed. The output is started by the InvalidateRect function.

The same functions are used with different parameters for the other three messages: WM\_KEYDOWN, WM\_KEYUP, and WM\_CHAR. The wsprintf function formats and saves characters in a buffer. (In this example each message has its own buffer.) wsprintf processes the text with the value contained in Param so that the text can be displayed later. Each message has its own Boolean variable, which is set to TRUE when the message is being processed. When the message is output, this variable is evaluated and deleted.

## Mouse input

You can use a mouse with almost all Windows applications. Windows supports mice with one, two, or three buttons. To determine whether a mouse is present, use the GetSystemMetrics function with the SM\_MOUSEPRESENT parameter. If the return value is TRUE, a mouse is installed.

It's also possible to display a mouse cursor (pointer) when a hardware mouse isn't present. Windows has an internal counter for the cursor that determines whether the cursor is displayed. The mouse cursor is visible when this counter is greater than or equal to zero.

If a mouse is installed, the counter is assigned an initialization value of 0; otherwise the value is -1. The ShowCursor function can increment or decrement the counter.

```
int ShowCursor( bShow )
```

bShow:      If this parameter is TRUE, "Display number" is incremented. If the parameter is FALSE, "Display number" is decremented.

int:      The new "Display number" is sent back.

Since the cursor is used by all Windows applications, it should be set to its original state before leaving the client area or before the window loses its focus.

## Mouse messages

Once the user moves the mouse cursor (pointer), Windows sends the WM\_MOUSEMOVE message. Each mouse button also has three messages that declare whether the mouse button is being pressed or released. There isn't an equivalent to the WM\_CHAR message.

However, you can prompt for a double-click. This occurs when the user quickly presses, releases, and then presses the mouse button. This process must occur within the defined double-click time of the system.

Message	Event
WM_MOUSEMOVE	Mouse moved in window
WM_LBUTTONDOWN	Left mouse button pressed
WM_LBUTTONUP	Left mouse button released
WM_MBUTTONDOWNDBCLK	Left mouse button double-clicked
WM_MBUTTONDOWN	Center mouse button pressed
WM_MBUTTONUP	Center mouse button released
WM_MBUTTONDOWNDBCLK	Center mouse button double-clicked
WM_RBUTTONDOWN	Right mouse button pressed
WM_RBUTTONUP	Right mouse button released
WM_RBUTTONDOWNDBCLK	Right mouse button double-clicked

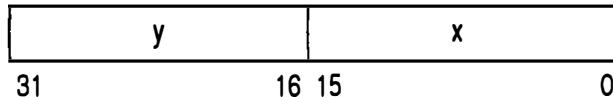
In addition to these mouse messages, there are eleven other messages that relate to the message area and are usually ignored in the window procedure. All of these messages begin with WM\_NC.

The parameters wParam and lParam have the same structure for all the mouse messages listed above. wParam contains a bit mask that specifies the current status of the mouse buttons, the **Shift** key, and the **Ctrl** key. lParam contains the x and y coordinates of the mouse pointer, which always refer to the upper-left corner of the client area. Use the LOWORD and HIWORD macros to obtain the x or y value.

## wParam:

Bitmask	Meaning
MK_CONTROL	Set if CONTROL key is pressed
MK_LBUTTON	Set if left mouse button is pressed
MK_MBUTTON	Set if center mouse button is pressed
MK_RBUTTON	Set if right mouse button is pressed
MK_SHIFT	Set If SHIFT key is pressed

## IParam:



To change the position of the cursor, use the SetCursorPos function. The x and y coordinates of the new position refer to the upper-left corner of the entire screen instead of to the client area. Use the ClientToScreen and ScreenToClient functions to transfer a point from one coordinate system to the other.

Windows always sends the mouse messages to the window that has the mouse capture set. Either the mouse pointer is in the window or the SetCapture function sets the mouse capture. With the second option, negative values can be used as x and y coordinates in IParam because the reference (upper-left corner of the client area) doesn't change.

SetCapture is normally used during important operations, such as loading an application. Since the mouse is present only once in the entire system, you should release the mouse capture, with the ReleaseCapture function, as soon as the operation is finished. Otherwise, other applications won't be able to use the mouse.

You also wouldn't be able to exit the current application by clicking on the system box because there wouldn't be any mouse messages for the non-client area. However, you would still need the WM\_NCLBUTTONDOWNDBLCLK message to exit the application.

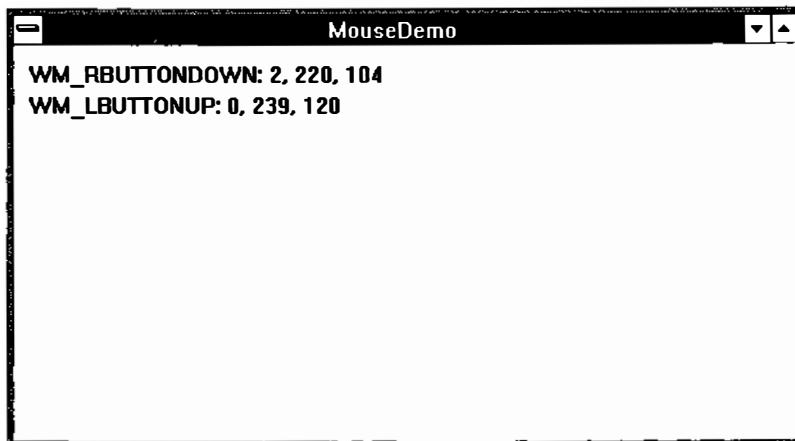
Windows will only send the double-click messages to the window procedure if the style parameter also received the CS\_DBLCLKS flag while the WNDCLASS structure was being furnished.

You can change the double-click time with the SetDoubleClickTime function. The new double-click time is then valid for the entire system, not only the particular application.

The following is the sequence of messages for double-clicking the left mouse button:

<b>Sequence</b>	<b>Message</b>
1st message	WM_LBUTTONDOWN
2nd message	WM_LBUTTONUP
3rd message	WM_LBUTTONDOWNDBLCLK
4th message	WM_LBUTTONUP

## Mouse example



This example displays text (similar to the text in the keyboard example) when you press the right mouse button and release the left mouse button. Each WM\_MOUSEMOVE message instructs the computer to beep. When you double-click the left mouse button, the Mouse capture is set and released.

### New messages

WM\_MOUSEMOVE  
WM\_RBUTTONDOWN  
WM\_LBUTTONUP  
WM\_LBUTTONDOWNDBLCLK

### Brief description

Sent when the mouse is moved  
Sent when the right mouse button is pressed  
Sent when the left mouse button is released  
Sent when the left mouse button is double-clicked

New functions	Brief description
SetCapture	All mouse input sent to the specified window
Releasecapture	Mouse capture is released
MessageBeep	Produces beep
New Macros	Brief description
HIGHWORD	Retrieves the high value word of a 'long integer' value
LOWORD	Retrieves the low value word of a 'long integer' value

## Source code: MOUSE\_BT.C

```
/** MOUSE_BT.C ****
/** Reads mouse button status and cursor position using WM_RBUTTONDOWN,    */
/** WM_LBUTTONUP and WM_MOUSEMOVE. Button status and cursor position at    */
/** the time of that status is displayed in the window. Beeping indicates   */
/** mouse movement.                                                       */
*****
```

```
#include "windows.h"                                // Include windows.h header file
#include "string.h"                                 // Include string.h header file

BOOL MouseInit ( HANDLE );                         // Mouse initialization
long FAR PASCAL MouseWndProc( HWND, unsigned, WORD, LONG);

/** WinMain (main function for every Windows application) ****

int PASCAL WinMain( hInstance, hPrevInstance, lpszCmdLine, cmdShow )
    HANDLE hInstance, hPrevInstance;      // Current & previous instances
    LPSTR lpszCmdLine;                  // Long ptr to string after
                                         // program name during execution
    int cmdShow;                       // Specifies the application
                                         // window's appearance
{
    MSG msg;                          // Message variable
    HWND hWnd;                        // Window handle

    if (!hPrevInstance)                // Initialize first instance
    {
        if (!MouseInit( hInstance )) // If initialization fails
            return FALSE;           // return FALSE
    }
}

/** Specify appearance of application's main window ****

hWnd = CreateWindow("MouseDemo",           // Window class name
```

```

        "MouseDemo",           // Window caption
        WS_OVERLAPPEDWINDOW, // Overlapped window style
        CW_USEDEFAULT,       // Default upper-left x pos.
        0,                  // Upper-left y pos.
        CW_USEDEFAULT,       // Default initial x size
        0,                  // y size
        NULL,               // No parent window
        NULL,               // Window menu used
        hInstance,           // Application instance
        NULL);              // No creation parameters

ShowWindow( hWnd, cmdShow );           // Make window visible
UpdateWindow( hWnd );                // Update window

while (GetMessage(&msg, NULL, 0, 0)) // Message reading
{
    TranslateMessage(&msg);          // Message translation
    DispatchMessage(&msg);          // Send message to Windows
}
return (int)msg.wParam;               // Return wParam of last message
}

BOOL MouseInit( hInstance )           // Mouse status
HANDLE hInstance;                   // Instance handle
{

/** Specify window class *****/
WNDCLASS      wcMouseClass;           // Main window class

wcMouseClass.hCursor      = LoadCursor( NULL, IDC_ARROW );           // Mouse cursor
wcMouseClass.hIcon        = LoadIcon( NULL, IDI_APPLICATION );        // Default icon
wcMouseClass.lpszMenuName = NULL;                                         // No menu
wcMouseClass.lpszClassName = "MouseDemo";                                // Window class
wcMouseClass.hbrBackground = GetStockObject( WHITE_BRUSH );            // White background
wcMouseClass.hInstance     = hInstance;                                    // Instance
wcMouseClass.style         = CS_VREDRAW | CS_HREDRAW | CS_DBLCLKS;      // Horizontal and vertical redraw of client area,
//                                                               and check for double-clicks
wcMouseClass.lpfnWndProc   = MouseWndProc;                            // Window function
wcMouseClass.cbClsExtra    = 0 ;                                         // No extra bytes
wcMouseClass.cbWndExtra    = 0 ;                                         // No extra bytes

if (!RegisterClass( &wcMouseClass ) )           // Register window class
    return FALSE;                      // Return FALSE if registration fails
}

```

## Keyboard and Mouse Input

---

```
// If registration is successful,
    return TRUE;                                // Return TRUE
}

/** MouseWndProc *****/
/** Main window function: All messages are sent to this window      */
//********************************************************************

long FAR PASCAL MouseWndProc( hWnd, message, wParam, lParam )
    HWND      hWnd;                           // Window handle
    unsigned   message;                      // Message type
    WORD      wParam;                        // Message-dependent 16 bit value
    LONG      lParam;                        // Message-dependent 32 bit value
{
    static char    chTextDown[35];           // Declare TextDown 35 characters
    static char    chTextUp[35];            // Declare TextUp 35 characters
    static BOOL    bCapture = FALSE;         // Set Capture default to FALSE
    HDC          hDC;                      // Device context handle
    PAINTSTRUCT  ps;                      // PAINTSTRUCT data structure

    ** Check button and movement status, display on screen ****

    switch (message)                      // Process messages
    {
        case WM_RBUTTONDOWN:             // Process if r. button pressed
            wsprintf(chTextDown, "WM_RBUTTONDOWN: %x, %d, %d",
                       wParam, LOWORD(lParam), HIWORD(lParam));
            InvalidateRect(hWnd, NULL, TRUE); // Redraw client area
            break;                      // End of this message process

        case WM_LBUTTONUP:              // Process if l. button released
            wsprintf(chTextUp, "WM_LBUTTONUP: %x, %d, %d",
                       wParam, LOWORD(lParam), HIWORD(lParam));
            InvalidateRect(hWnd, NULL, TRUE); // Redraw client area
            break;                      // End of this message process

        case WM_LBUTTONDOWNDBLCLK:       // Double-click
            if (bCapture)               // bCapture sent?
            {
                ReleaseCapture();     // Release mouse capture &
                bCapture = FALSE;      // set bCapture to FALSE
            }
            else                      // If no bCapture sent
            {
                SetCapture(hWnd);     // Set mouse capture &
                bCapture = TRUE;       // set bCapture to TRUE
            }
    }
}
```

```

        break;                                // End of this message process

    case WM_MOUSEMOVE:                      // Beep during mouse movement
        MessageBeep(0);                    // Sound beep
        break;                                // End of this message process

    case WM_PAINT:                          // Redraw client area window
        hDC = BeginPaint (hWnd, &ps); // Begin WM_PAINT process
                                            // Display text...
        TextOut (hDC, 10, 10, chTextDown, strlen(chTextDown));
        TextOut (hDC, 10, 30, chTextUp, strlen(chTextUp));
        EndPaint (hWnd, &ps);                //...and end WM_PAINT
        break;                                // End of this message process

    case WM_DESTROY:                        // Send WM_QUIT if window is
        PostQuitMessage (0); // destroyed
        break;                                // End of this message process
    default:                               // Send other messages to
                                            // default window function
        return (DefWindowProc( hWnd, message, wParam, lParam ));
        break;                                // End of this message process
    }
    return (0L);
}
}

```

## Module definition file: MOUSE\_BT.DEF

NAME	MouseDemo
DESCRIPTION	'Mouse input example'
EXETYPE WINDOWS	
STUB	'WINSTUB.EXE'
CODE	PRELOAD MOVEABLE
DATA	PRELOAD MOVEABLE MULTIPLE
HEAPSIZE	4096
STACKSIZE	4096
EXPORTS	MouseWndProc @1

To compile and link this application, use the COMPILE.BAT batch file described earlier in this book. You can create this file using the DOS

COPY CON command or any text editor or word processor that generates ASCII files. Here's the listing again:

```
cl -c -Gw -Zp %1  
link /align:16 %1,%1.exe,,libw+slibcew,%1.def  
rc %1.exe
```

Save this file to a directory contained in your path. Then when you are in the directory containing your source and definition files, type the following and press **Enter**:

```
compile mouse_bt
```

## **How MOUSE\_BT.EXE works**

The programming logic for the WM\_RBUTTONDOWN and WM\_LBUTTONUP messages, including the output, can be accepted. Only the Boolean variables are omitted. The value of lParam, which is divided with the HIWORD and LOWORD macros to obtain the x and y coordinates, is also output.

As soon as the window is set to capture and the mouse is moved, the WM\_MOUSEMOVE messages occur. The MessageBeep function responds to these messages. This indicates that this message appears frequently. The application message queue contains only one WM\_MOUSEMOVE message at a time. The style parameter in the WNDCLASS structure must be expanded for any WM\_LBUTTONDOWNDBLCLK messages to appear.

```
wcMouseClass.style = ... | CS_DBCLK
```

When this message is being processed, a Boolean variable is set. This variable indicates whether the mouse capture is currently set. The SetCapture function, as the only parameter, must receive the handle of the window that will contain the mouse capture. The ReleaseCapture function doesn't need any parameters.

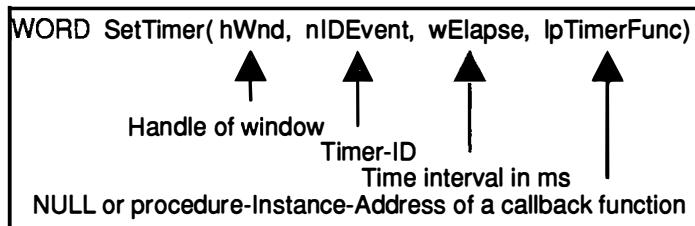
# Timer input

The Windows timer is another input device that informs an application when the previously set time interval has expired. The SYSTEM.DRV driver is responsible for hardware timer interrupts in Windows. The timer can be used for various tasks. In addition to clock applications, the timer can increase the efficiency of the multitasking environment for large tasks.

Each application should return the control of the CPU to Windows as quickly as possible. Therefore, long processes are often divided into individual sections. During a WM\_TIMER message, only one section is processed at a time. If you're working with the parallel or serial port you also need WM\_TIMER messages. Since the port doesn't produce its own messages, the WM\_TIMER messages must be used to prompt the port for data.

## Timer messages

Sixteen timers can be active simultaneously in the Windows system. Use the SetTimer function to start a timer. Then, depending on the time you specified, a WM\_TIMER message is generated. If the function retrieves a NULL, all the timers are currently assigned and a message isn't produced. Each application that's working with a timer should prompt for the return value and react accordingly. SetTimer has four parameters:



The third parameter specifies the time interval in milliseconds. Since the timer is dependent on the hardware timer, WM\_TIMER messages cannot occur more than 18.2 times per second. It's possible that the message won't go to the window procedure exactly after the specified time interval.

This can be caused by delays, for example from another application that uses the CPU longer. Also, since the WM\_TIMER message has a low priority in the application message queue and there can be only one WM\_TIMER message in this queue at a time, messages can be lost if they aren't removed from the queue promptly.

The fourth parameter is often set to NULL. In this case the WM\_TIMER messages are passed to the normal window function. You could also include the address of a callback function. This would enable the callback function, instead of the window procedure, to receive the WM\_TIMER messages. First you must determine the necessary address by using the MakeProcInstance function. We'll discuss this function in the chapter on dialog boxes resources.

With a WM\_TIMER message, the ID value of the timer, through which the message was produced, is in wParam. The lParam parameter isn't used.

The set timer runs until it's stopped by the KillTimer function. Since more than one timer can be started for an application, the ID value of the timer must be given to this function. Each WM\_TIMER message that belongs to this timer is removed from the message queue by the KillTimer function.

## Timer example

This application attempts to start two timers. If the application is successful in starting the timers, it then displays a window. If Windows already has its maximum number of timers running, a message box informs the user of this fact. The user then has the option of retrying a timer startup, or cancelling the application.

Once the timers are active, one timer beeps about once per second, while the other beeps twice about every five seconds. These times are only approximate, and vary with two factors:

- 1) Your PC's processing speed.
- 2) The number of applications open in addition to TIMER.EXE.

---

New messages	Brief description
WM_TIMER	Sent after set interval of time expires
New functions	Brief description
SetTimer	Produces a system timer event
KillTimer	Deletes the specified timer event
MessageBox	Creates a window with the specified text

## Source code: TIMER.C

```
/** TIMER.C ****
/** Starts two timers which instruct the computer to beep at one second & */
/** approx. five second intervals respectively. Intervals vary with the */
/** number of applications active!
****

#include "windows.h"                                // Include windows.h header file

#define SEC_1    1                                  // Define first SEC_ variable
#define SEC_5    2                                  // Define second SEC_ variable

BOOL TimerInit ( HANDLE );                         // Timer initialization
long FAR PASCAL TimerWndProc( HWND, unsigned, WORD, LONG);

/** WinMain (main function for every Windows application) ****

int PASCAL WinMain( hInstance, hPrevInstance, lpszCmdLine, cmdShow )
    HANDLE hInstance, hPrevInstance;      // Current & previous instances
    LPTSTR lpszCmdLine;                  // Long ptr to string after
                                         // program name during execution
    int     cmdShow;                    // Specifies the application
                                         // window's appearance

{
MSG msg;                                         // Message variable
HWND hWnd;                                       // Window handle

    if (!hPrevInstance)                      // Initialize first instance
    {
        if (!TimerInit( hInstance )) // If initialization fails
            return FALSE;                // return FALSE
    }

/** Specify appearance of application's main window ****
```

## Keyboard and Mouse Input

---

```
hWnd = CreateWindow("Timer",           // Window class name
                    "Timer",           // Window caption
                    WS_OVERLAPPEDWINDOW, // Overlapped window style
                    CW_USEDEFAULT,     // Default upper-left x pos.
                    0,                 // Upper-left y pos.
                    CW_USEDEFAULT,     // Default initial x size
                    0,                 // y size
                    NULL,              // No parent window
                    NULL,              // Window menu used
                    hInstance,          // Application instance
                    NULL);             // No creation parameters

while (!SetTimer(hWnd, SEC_1, 1000, NULL)) // Too many
                                              // timers (16) running?
    if (IDCANCEL == MessageBox(hWnd, "2 timers too many started",
                               "Timer", MB_RETRYCANCEL)) // ERROR
        return FALSE;                      // Return FALSE

while (!SetTimer(hWnd, SEC_5, 5000, NULL)) // Too many
                                              // timers (16) running?
    if (IDCANCEL == MessageBox(hWnd, "1 timer too many started",
                               "Timer", MB_RETRYCANCEL)) // ERROR
        return FALSE;                      // Return FALSE

ShowWindow( hWnd, cmdShow );           // Make window visible
UpdateWindow( hWnd );                // Update window

while (GetMessage(&msg, NULL, 0, 0)) // Message reading
{
    TranslateMessage(&msg);           // Message translation
    DispatchMessage(&msg);           // Send message to Windows
}

return (int)msg.wParam;               // Return wParam of last message
}

BOOL TimerInit( hInstance )           // Timer status
HANDLE hInstance;                   // Instance handle
{

/** Specify window class ****
WNDCLASS      wcTimerClass;

wcTimerClass.hCursor      = LoadCursor( NULL, IDC_ARROW );
wcTimerClass.hIcon        = LoadIcon( NULL, IDI_APPLICATION );
wcTimerClass.lpszMenuName = NULL;
wcTimerClass.lpszClassName = "Timer";
```

```

wcTimerClass.hbrBackground      = GetStockObject( WHITE_BRUSH );
wcTimerClass.hInstance          = hInstance;
wcTimerClass.style              = CS_VREDRAW | CS_HREDRAW;
wcTimerClass.lpfnWndProc        = TimerWndProc;
wcTimerClass.cbClsExtra         = 0 ;
wcTimerClass.cbWndExtra         = 0 ;

if (!RegisterClass( &wcTimerClass ) )
    return FALSE;

return TRUE;
}

/** TimerWndProc *****/
/** Main window function: All messages are sent to this window ***/
/** *****/

long FAR PASCAL TimerWndProc( hWnd, message, wParam, lParam )
    HWND           hWnd;
    unsigned        message;
    WORD           wParam;
    LONG           lParam;
{

/** Perform timer task *****/

switch (message)
{
    case WM_TIMER:
        if (wParam == SEC_1) // Beep once a second
            MessageBeep(0);
        if (wParam == SEC_5) // Beep once every five seconds
        {
            MessageBeep(0);
            MessageBeep(0);
        }
        break;

    case WM_DESTROY:           // Destroy window
        KillTimer(hWnd, SEC_1);
        KillTimer(hWnd, SEC_5);
        PostQuitMessage(0);
        break;

    default:
        return (DefWindowProc( hWnd, message, wParam, lParam ));
        break;
}

```

```
    return(0L);  
}
```

## Module definition file: TIMER.DEF

NAME	Timer
DESCRIPTION	'Timer example'
EXETYPE	WINDOWS
STUB	'WINSTUB.EXE'
CODE	PRELOAD MOVEABLE
DATA	PRELOAD MOVEABLE MULTIPLE
HEAPSIZE	4096
STACKSIZE	4096
EXPORTS	TimerWndProc @1

To compile and link this application, use the COMPILE.BAT batch file described earlier in this book. You can create this file using the DOS COPY CON command, any text editor or word processor that generates ASCII files. Here's the listing again:

```
cl -c -Gw -Zp %1  
link /align:16 %1,%1.exe,,libw+slibcew,%1.def  
rc %1.exe
```

Save this file to a directory contained in your path. Then when you are in the directory containing your source and definition files, type the following and press **Enter**:

```
compile timer
```

The batch file performs all the tasks needed.

## How TIMER.EXE works

In this application the main procedure WinMain is expanded for the first time. Before the window appears, the application tries to start the two timers with the SetTimer function. Since the first parameter is the

handle of the window, the function reference must occur after CreateWindow.

The second parameter is the ID value of the timer, which was defined at the beginning of the application. The time is specified in milliseconds.

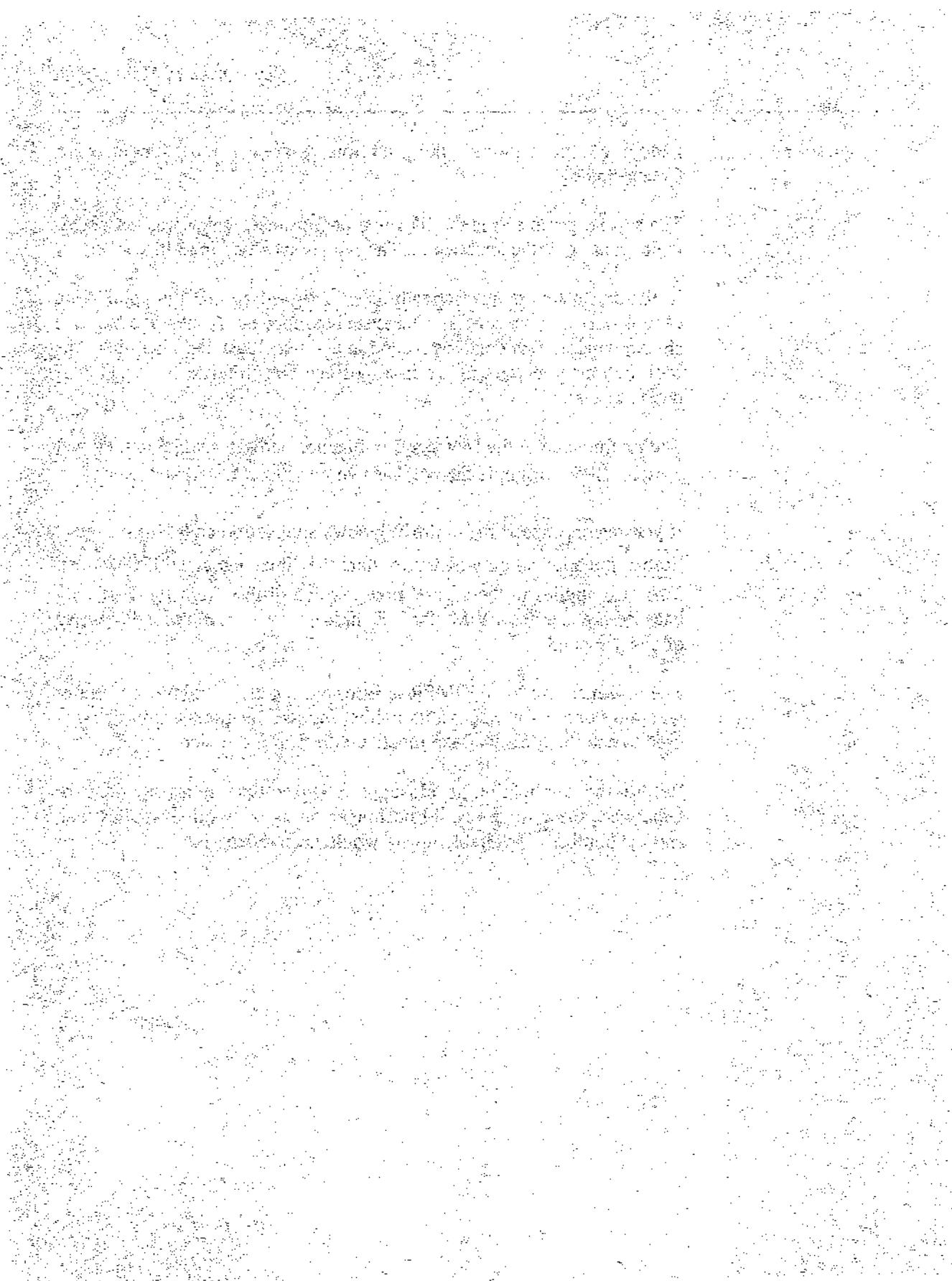
If all sixteen timers have already been assigned so that the application cannot start the two timers, a message box appears. A message box is a simple window that contains a caption and text. You can also select the push buttons you want to use in a message box (refer to the chapter on dialog boxes).

The return value of the MessageBox function indicates which button was pressed. In our example the box has two buttons: **Cancel** and **Retry**.

If you select **Cancel** the entire Windows application ends. If you select **Retry** the application will try to start the timer again with SetTimer. This also applies to the second timer, which differs from the first only because it generates a WM\_TIMER message every five seconds instead of every second.

The new message WM\_TIMER is inserted into the window procedure. wParam contains the ID value of the appropriate timer. This timer is used to call the MessageBeep function either once or twice.

Before the application is closed the two timers must be deleted. Otherwise, these timers could no longer be used by other applications and the number of available timers would keep decreasing.



# Output

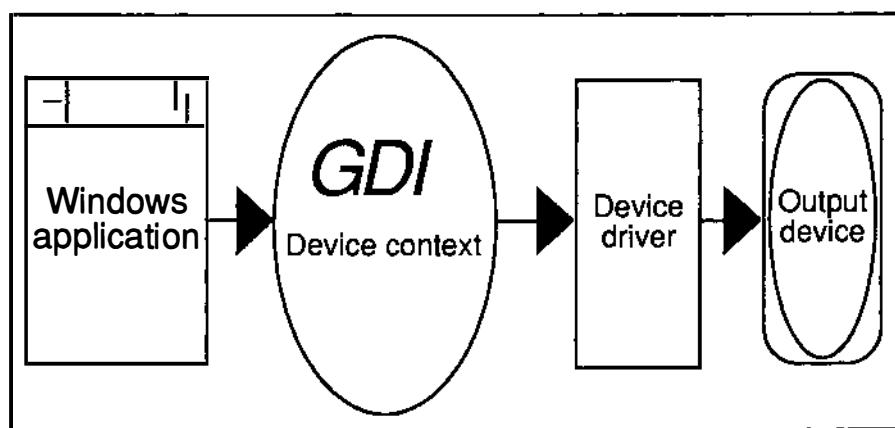
## Overview

In this chapter we'll examine how the Windows environment produces output. We'll discuss display context, pens and brushes, and the WM\_PAINT message. Also, we'll list some output functions.

The Graphics Device Interface (GDI) is responsible for all output and the Windows DLL GDI.EXE contains all of the graphic functions.

## Device context

You must have a device context to produce output in Windows.



A device context is the connection between a Windows application, a device driver, and an output device such as a monitor. Each application in Windows can use the GDI functions to produce output on an output device.

The GDI is responsible for passing the references, which are device-independent, to the device driver. The driver then translates these GDI references into device-dependent operations.

## Device context attributes

Therefore, the process of producing output always begins by obtaining a device context. There are various functions you can use to do this. We'll discuss these functions in more detail later.

You retrieve a handle to the device context, which you need as a parameter for almost all of the subsequent Graphics Device Interface (GDI) functions. Windows uses the device context to save certain attributes. So a device context contains a certain data structure, which cannot be directly accessed. You must use GDI functions to change individual attributes. A new device context is filled with default values.

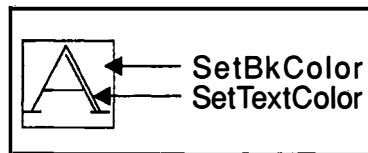
Attribute	Default	GDI functions
Background Color	White	SetBkColor
Background Mode	OPAQUE	SetBkMode
Bitmap	no default	CreateBitmap CreateBitmapIndirect CreateCompatibleBitmap SelectObject
Brush	WHITE_BRUSH	CreateBrushIndirect CreateDIBPatternBrush CreateHatchBrush CreatePatternBrush CreateSolidBrush SelectObject
Brush Origin	(0,0)	SetBrushOrg UnrealizeObject
Clipping Region	Display interface	ExcludeClipRect IntersectClipRect OffsetClipRgn SelectClipRgn
Color Palette	DEFAULT_PALETTE	CreatePalette RealizePalette SelectPalette
Current Pen Position	(0,0)	MoveTo
Drawing Mode	R2_COPYPEN	SetROP2
Font	SYSTEM_FONT	CreateFont CreateFontIndirect
Intercharacter Spacing	0	SelectObject
Mapping Mode	MM_TEXT	SetTextCharacterExtra
Pen	BLACK_PEN	SetMapMode CreatePen CreatePenIndirect SelectObject
Polygon-filling Mode	ALTERNATE	SetPolyFillMode
Relative-absolute Flag	ABSOLUTE	SetRelAbs
Stretching Mode	BLACKONWHITE	SetStretchBltMode

Text Color	Black	SetTextColor
Viewport Extent	(1,1)	SetViewportExt
Viewport Origin	(0,0)	SetViewportOrg
Window Extent	(1,1)	SetWindowExt
Window Origin	(0,0)	SetWindowOrg

The background color attribute in the device context isn't the same background that's defined in the WNDCLASS structure. The background in the window class is a brush (i.e., a pattern that could also be monochrome).

If necessary, Windows uses this brush to erase the client area. (This deletion is caused by the WM\_ERASEBKGND message in DefWindowProc.) However, the brush isn't part of the device context.

To change many of the settings of the WNDCLASS structure, including the brush for the background, use the SetClassWord function. When the background mode is set to OPAQUE, the background color fills the space between letters and lines. When the background mode is set to TRANSPARENT, the background color is ignored (SetBkColor, SetBkMode).



## Device context types

There are several types of device contexts. The type of device context that's used depends on the output device to which it's assigned. A display context always refers to either an entire window or sections of a window. Device contexts apply to an entire device (e.g., a printer or the entire screen).

The memory device context, which is always a copy of a real device context, is located in memory. This device context is used when you're working with bitmaps (refer to the chapter on bitmaps). Each type of device context has its own function pair, which the device context creates and deletes.

<b>Device context type</b>	<b>Function pair for creation and deletion</b>
Clipping region	BeginPaint --> EndPaint
Entire client area	GetDC --> ReleaseDC
Entire window	GetWindowDC --> ReleaseDC
Device DC	CreateDC --> DeleteDC
Memory DC	CreateCompatibleDC --> DeleteDC

With the device context for the entire window it's also possible to paint the title bar, etc., through the system menu. Since this function was created for certain paint effects, we don't recommend using it for normal drawing.

## Display context types

In addition to the display context for the entire window, there are three other display contexts: common, class, and private. These display contexts produce output only in the client area.

The common device context is the default device context for all windows and is the most frequently used device context. When you call BeginPaint or GetDC, a device context, along with its handle, will be retrieved from a cache of five display contexts. You can use this handle to set the attributes we mentioned above and produce the desired output. Once the output is complete, you must use EndPaint or ReleaseDC to return the display context to the cache. All of the set attributes will be lost.

After the context is released, no output will appear until you obtain the next device context. You should obtain and release a common device context only if you're processing one message instead of several. This is important because you can't be sure that the message that takes place when you release the device context will actually occur.

If you don't release the common device context and the other four device contexts from the cache, Windows will ignore all subsequent requests for a device context. This cache doesn't apply to the class device context or the private device context. Similar to the double-click mouse message, both of these device context types are specified in the style parameter of the WNDCLASS structure: CS\_CLASSDC and CS\_OWNDC.

A class device context is used by all windows that relate to the same class with the CS\_CLASSDC style, even if these windows belong to different

instances. You must obtain this device context before working with it. However, you don't have to release this device context to the cache afterwards because it didn't originate from there.

If only one window is working with the class display context, the set attributes will be preserved because the EndPaint and ReleaseDC functions don't have any effect. If another window of the same class also wants to access this device context, it must first obtain its own handle to the device context with BeginPaint or GetDC.

Except for the clipping region and the coordinate region, all the attributes Windows sets for the new window remain unchanged in the device context. Along with the other attributes, the coordinate origin is in a display context. It specifies the coordinates of the upper-left corner of the window in relation to the upper-left corner of the screen.

When a window changes an attribute (e.g., the color of the text) in the class device context, the change affects each window that subsequently uses the device context. Obviously this can produce unwanted effects.

A private display context is allocated to one window. The class to which the window belongs must obtain the CS\_OWNDC style. This device context must be created only once—it exists until the window closes. Both functions for releasing a common device context don't affect a private device context.

A disadvantage of the private device context is that Windows requires 800 bytes for each device context it saves.

Use the SaveDC function to save the attributes of a common display context so that you can set and use other attributes. Later you can use the RestoreDC function to retrieve the old settings. However, you cannot release the display context in between the two functions.

## The WM\_PAINT message

In Windows applications you should produce output only with the WM\_PAINT message.

### Origin

There are several ways a WM\_PAINT message can occur. Whenever Windows determines that the contents of the client area are no longer up-to-date, the system produces a WM\_PAINT message and places it in the application message queue. Frequently the entire client area is valid except for a rectangular area called the clipping region.

Windows allows you to draw only within the clipping region. For example, if you try to output text in the protected (still valid) part of the client area, the text won't appear. Therefore, Windows guarantees that current areas aren't overwritten. The invalid area is retrieved with a WM\_PAINT message and can be tested.

One way the window procedure can obtain a WM\_PAINT message is if part of the window had been under another window and is now in the foreground.

Another way the message could be obtained is if the user changed the size of the window and the two style parameters CS\_HREDRAW and CS\_VREDRAW were specified in the WNDCLASS structure. If these parameters aren't set, a WM\_PAINT message is generated only when the window is enlarged.

In a few other instances Windows tries to save an area of the display itself in order to restore it later. However, Windows sends a WM\_PAINT message if it fails to save the area. For example, this would happen if Windows removes a dialog box that was overlapping part of a window or if a menu is pulled down.

There are other instances when Windows saves the display area that's overwritten and then restores it. For example, when the cursor is moved across the client area.

The programmer can also produce a WM\_PAINT message by using certain functions such as the UpdateWindow function. It is called in

WinMain before the message loop and generates the first WM\_PAINT message. Windows immediately passes this message to the window procedure as a nonqueued message. This is how the client area is painted with the brush, which is a background brush in the WNDCLASS structure.

As we mentioned earlier, all output should be made when WM\_PAINT is being processed. However if something happens, such as output encountering a keypress, you need a function that can generate a WM\_PAINT message at any time. The InvalidateRect and InvalidateRgn functions are used to do this.

```
InvalidateRect ( hWnd, lpRect, bErase )
```

- |         |   |
|---------|---|
| hWnd:   | Identifies the window that should be modified.  |
| lpRect: | Specifies a rectangle structure containing either the clipping region, or NULL if the entire client area should be redrawn. |
| bErase: | If this parameter is TRUE, the background is erased before new output; if it is FALSE, the background is preserved.         |

## Processing messages

The WM\_PAINT message is always the last message in the application message queue. All other messages are retrieved from the queue first with GetMessage. This prevents the window contents from flickering.

There is always one WM\_PAINT message in the queue. If the last message hasn't been processed and a new message arrives, a new invalid rectangle is created out of the supplied coordinates of the old and new clipping region.

The two style parameters CS\_VREDRAW (vertical redraw) and CS\_HREDRAW (horizontal redraw), which are set in the WNDCLASS structure, determine the following: Whether a message is generated when a window is being resized and whether the entire client area or only the new part is set to invalid.

If CS\_VREDRAW or CS\_HREDRAW is set, when the user resizes the window, both a WM\_SIZE and WM\_PAINT message are produced and the entire client area is invalid. If only the window is enlarged and the new part of the client area is invalid, only a WM\_PAINT message is generated.

If output is produced with WM\_PAINT you must use the function pair BeginPaint and EndPaint to obtain and release a display context.

The BeginPaint function contains the parameters hWnd and &ps. hWnd is the handle of the window, whose client area should be either entirely or partially repainted. The &ps parameter represents the address of a variable of PAINTSTRUCT. When you call the function, this variable is filled with current values.

## **Paintstruct:**

<b>Field</b>	<b>Description</b>
hdc	Device context handle
fErase	TRUE = Background is erased
rcPaint	Rectangle structure of the clipping region
fRestore	Field reserved by Windows
flncUpdate	Field reserved by Windows
fgbReserved[16]	Fields reserved by Windows

BeginPaint:

The handle of the device is located in both the first parameter and the return value of the BeginPaint function. Usually the fErase flag is set to TRUE so that the background is erased before the new output. To do this, a WM\_ERASEBKGND message is generated by Windows.

The WM\_ERASEBKGND message is usually passed to the default window procedure. This causes the background to be painted over with the brush specified in the WNDCLASS structure. The rectangle structure is of the RECT data type and defines the upper-left and lower-right corner of the invalid area in relation to the upper-left corner of the client area.

---

**EndPaint:** The handle of the device context is released again and the invalid area is set to valid (i.e., the contents of the client area are current).

If the WM\_PAINT message is passed to DefWindowProc, the following occurs:

```
case WM_PAINT:  
BeginPaint(hWnd, &ps);  
EndPaint(hWnd, &ps);  
break;
```

If a function isn't called between BeginPaint and EndPaint, the rectangle that was previously invalid is set. Also, Windows sends another WM\_ERASEBKGND message which erases the background. All output from messages other than WM\_PAINT is erased also.

You cannot change this by answering the WM\_PAINT message with the break statement. Since EndPaint isn't being called, the invalid area isn't set to valid. Windows then sends the WM\_PAINT message again. As a result, the application and perhaps even the entire system could hang up.

## Output functions

The simpler output functions are divided into the basic text and graphic function groups. All functions, except LineDDA, have the first parameter (the handle of the device context hDC) in common. We'll discuss bitmap functions, which also produce output, in a separate chapter.

## Text functions

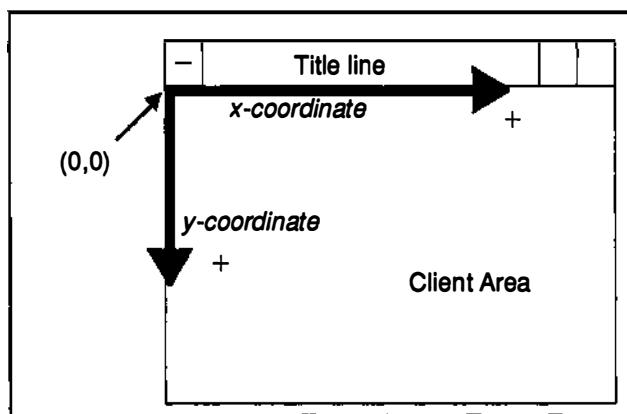
TextOut and DrawText are the most frequently used text functions for producing output. We'll discuss these in more detail later. There are also other text output functions that have a more complicated structure (e.g., GrayString).

```
BOOL TextOut(hDC, X, Y, lpString, nCount)
```

## Output

Parameter	Data type	Description
hDC	HDC	Handle of device context
X	int	x coordinate - start of text
Y	int	y coordinate - start of text
lpString	LPSTR	Pointer to text buffer
nCount	int	Number of characters

Unless a setting was previously made with the SetTextAlign function, the x and y coordinates specify, in logical coordinates, the point at which the text begins. If the default mapping mode (MM\_TEXT) is being used, the reference point is the upper-left corner of the client area.



The length of the string is necessary because the function doesn't recognize any character strings closed with NULL bytes. The text also shouldn't have any ASCII control characters, such as Linefeed or Tab, because they wouldn't be properly interpreted when they are output.

Also, the current pen position in the device context isn't changed by this function unless you previously specified a setting with the SetTextAlign function.

The keyboard and mouse sample programs use the TextOut function. The output for these examples is a "black box". Let's examine the main window function used by the MOUSE\_BT application described in the previous chapter:

```
/** MouseWndProc:*****  
 ** Main window function: All messages are sent to this window ****/  
*****  
  
long FAR PASCAL MouseWndProc( hWnd, message, wParam, lParam )  
{  
    HWND          hWnd;  
    unsigned      message;  
    WORD          wParam;  
    LONG          lParam;  
  
    static char   chTextDown[35];  
    static char   chTextUp[35];  
    static BOOL   bCapture = FALSE;  
/****** Output parameters *****/  
    HDC           hDC;  
    PAINTSTRUCT  ps;  
/******  
  
/** Check button and movement status, display on screen *****/  
  
    switch (message)  
    {  
        case WM_RBUTTONDOWN:          // Right button pressed  
            wsprintf(chTextDown, "WM_RBUTTONDOWN: %x, %d, %d",  
                      wParam, LOWORD(lParam), HIWORD(lParam));  
            InvalidateRect(hWnd, NULL, TRUE);  
            break;  
  
        case WM_LBUTTONUP:           // Left button released  
            wsprintf(chTextUp, "WM_LBUTTONUP: %x, %d, %d",  
                      wParam, LOWORD(lParam), HIWORD(lParam));  
/****** Generate WM_PAINT *****/  
            InvalidateRect(hWnd, NULL, TRUE);  
/******  
            break;  
  
        case WM_LBUTTONDOWNDBLCLK:    // Double-click  
            if (bCapture)  
            {  
                ReleaseCapture();  
                bCapture = FALSE;  
            }  
            else  
            {  
                SetCapture(hWnd);  
                bCapture = TRUE;  
            }  
    }  
}
```

## Output

---

```
        break;

    case WM_MOUSEMOVE:           // Beep during mouse movement
        MessageBeep(0);
        break;

/********************* OUTPUT *****/
    case WM_PAINT:              // Redraw client area window
        hDC = BeginPaint (hWnd, &ps);
        TextOut (hDC, 10, 10, chTextDown, strlen(chTextDown));
        TextOut (hDC, 10, 30, chTextUp, strlen(chTextUp));
        EndPaint (hWnd, &ps);
        break;
/********************* */

    case WM_DESTROY:             // Destroy window
        PostQuitMessage(0);
        break;
    default:
        return (DefWindowProc( hWnd, message, wParam, lParam ));
        break;
    }
    return(0L);
}

}
```

Text should be output when the right mouse button is pressed. Instead of being produced immediately by WM\_RBUTTONDOWN, the output collides with the InvalidateRect function. This generates a WM\_PAINT message and the second parameter causes the entire client area to be marked invalid.

In the WM\_PAINT message, a handle to a display context is obtained with BeginPaint. Then the text can be output with TextOut. Finally, the EndPaint function releases the device context and sets the client area to valid.

```
int DrawText( hdc, lpString, nCount, lpRect, wFormat)
```

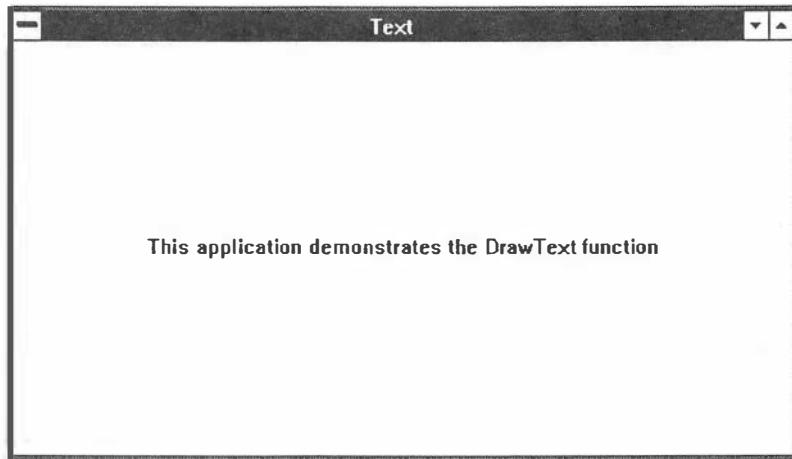
Parameter	Data type	Description
hDC	HDC	Handle of the device context
lpString	LPSTR	Pointer to the text buffer
nCount	int	Number of characters
lpRect	LPRECT	Pointer to a rectangle for formatting text
wFormat	WORD	Method of formatting

You can easily format text with the DrawText function. A pointer to a rectangular structure is passed as the fourth parameter. The text is written in this rectangle. The third parameter can contain the value -1 instead of the length for a string closed by a NULL byte. The value -1 specifies that the function should calculate the length of the text itself.

The last parameter contains a combination of different flags (e.g., a flag for an automatic word break, DT\_WORDBREAK) and one for displaying the text as centered within the rectangle (DT\_CENTER and DT\_VCENTER).

Unless the DT\_SINGLELINE flag is set, both the carriage return (13) and the linefeed (10) are interpreted as new lines.

## Example of the DrawText function



This sample application tests three cases in addition to the DrawText function. These cases apply to the style parameters CS\_HREDRAW and CS\_VREDRAW.

### New messages

WM\_PAINT

### Brief description

Sent after prompt to repaint

## *Output*

---

New functions	Brief description
BeginPaint	Prepares a window for repainting
EndPaint	Marks the end of painting
GetClientRect	Copies the coordinates of a client area
DrawText	Outputs a string
New structures	Brief description
PAINTSTRUCT	Contains character information
RECT	Rectangle structure

## **Source code: TEXT.C**

```
/** TEXT.C ****
/** Displays text in a window using the DrawText function ****
****

#include "windows.h"                                // Include windows.h header file

BOOL TextInit ( HANDLE );                           // Text initialization
long FAR PASCAL TextWndProc( HWND, unsigned, WORD, LONG);

/** WinMain (main function for every Windows application) *****

int PASCAL WinMain( hInstance, hPrevInstance, lpszCmdLine, cmdShow )
HANDLE hInstance, hPrevInstance;                   // Current & previous instances
LPSTR lpszCmdLine;                               // Long ptr to string after
                                                // program name during execution
int cmdShow;                                     // Specifies the application
                                                // window's appearance
{
    MSG msg;                                     // Message variable
    HWND hWnd;                                    // Window handle

    if (!hPrevInstance)                          // Initialize first instance
    {
        if (!TextInit( hInstance )) // If initialization fails
            return FALSE;                      // return FALSE
    }
}

/** Specify appearance of application's main window *****

hWnd = CreateWindow("Text",                  // Window class name
                    "Text",                  // Window caption
                    WS_OVERLAPPEDWINDOW, // Overlapped window style
                    CW_USEDEFAULT,       // Default upper-left x. pos
```

```
        0,                                // Upper-left y pos.
        CW_USEDEFAULT,                  // Default initial x size
        0,                                // y size
        NULL,                             // No parent window
        NULL,                             // Window menu used
        hInstance,                        // Application instance
        NULL);                           // No creation parameters

ShowWindow( hWnd, cmdShow );           // Make window visible
UpdateWindow( hWnd );                // Update window

while (GetMessage(&msg, NULL, 0, 0)) // Message reading
{
    TranslateMessage(&msg);          // Message translation
    DispatchMessage(&msg);          // Send message to Windows
}

return (int)msg.wParam;               // Return wParam of last message
}

BOOL TextInit( hInstance )           // Text initialization
HANDLE hInstance;
{
    /* Specify window class *****/
    WNDCLASS      wcTextClass;           // Main window class

    wcTextClass.hCursor     = LoadCursor( NULL, IDC_ARROW );
                                // Mouse cursor
    wcTextClass.hIcon       = LoadIcon( NULL, IDI_APPLICATION );
                                // Default icon
    wcTextClass.lpszMenuName = NULL;
                                // No menu
    wcTextClass.lpszClassName = "Text";
                                // Window class
    wcTextClass.hbrBackground = (HBRUSH)GetStockObject( WHITE_BRUSH );
                                // White background
    wcTextClass.hInstance   = hInstance;    // Instance
    wcTextClass.style       = CS_VREDRAW | CS_HREDRAW; // Horizontal and
                                                // vertical redraw of client area
    wcTextClass.lpfnWndProc = TextWndProc;    // Window function
    wcTextClass.cbClsExtra  = 0 ;
                                // No extra bytes
    wcTextClass.cbWndExtra  = 0 ;
                                // No extra bytes

    if (!RegisterClass( &wcTextClass ) )      // Register window class
        return FALSE;                      // Return FALSE if registration fails
                                            // If registration is successful,
                                            // return TRUE
}
```

## *Output*

---

```
/** TextWndProc ****
** Main window function: All messages are sent to this window      */
/****

long FAR PASCAL TextWndProc( hWnd, message, wParam, lParam )
HWND      hWnd;                      // Window handle
unsigned   message;                  // Message type
WORD       wParam;                  // Message-dependent 16 bit value
LONG      lParam;                  // Message-dependent 32 bit value
{

    HDC          hDC;                // Device context handle
    PAINTSTRUCT ps;               // PAINTSTRUCT data structure
    RECT         rect;              // RECT rectangle structure
    char        chText[] = "This application demonstrates
                           the DrawText function";

    switch (message)           // Process messages
    {

        case WM_PAINT:          // Client area redraw
            BeginPaint (hWnd, &ps); // Begin WM_PAINT process
            hDC = ps.hdc;
            GetClientRect(hWnd, &rect); // Get client area rectangle
            DrawText( hDC, chText, -1, &rect, DT_CENTER | DT_VCENTER |
                       DT_SINGLELINE);
                           // Draw text centered in rectangle
            EndPaint(hWnd, &ps); // End WM_PAINT process
            break;                // End of this message process

        case WM_DESTROY:         // Destroy window
            PostQuitMessage(0);
            break;                // End of this message process

        default:                 // Send other messages to default
                               // window function
            return (DefWindowProc( hWnd, message, wParam, lParam ));
            break;                // End of this message process
    }
    return(0L);
}
```

## **Module definition file: TEXT.DEF**

NAME	Text
------	------

---

```
DESCRIPTION  'DrawText example'

EXETYPE      WINDOWS

STUB         'WINSTUB.EXE'

CODE          PRELOAD MOVEABLE
DATA          PRELOAD MOVEABLE MULTIPLE

HEAPSIZE     4096
STACKSIZE    4096

EXPORTS      TextWndProc @1
```

To compile and link this application, use the COMPILE.BAT batch file described earlier in this book. You can create this file using the DOS COPY CON command or any text editor or word processor that generates ASCII files. Here's the listing again:

```
cl -c -Gw -Zp %1
link /align:16 %1,%1.exe,,libw+slibcew,%1.def
rc %1.exe
```

Save this file to a directory contained in your path. Then when you are in the directory containing your source and definition files, type the following and press **Enter**:

```
compile text
```

The batch file performs all the tasks needed.

## How TEXT.EXE works

In the first case the style parameters CS\_VREDRAW and CS\_HREDRAW are used and the rectangle, in which the text is supposed to be centered, is calculated with the GetClientRect function. This function determines the current size of the client area and places this value in a variable of the RECT structure. The upper-left and lower-right corner of the client area are defined in this structure. Since the reference point is the upper-left corner, the coordinates (0,0) are entered.

The text should be centered both vertically and horizontally. For the DT\_VCENTER flag to work, the DT\_SINGLELINE flag must also be specified.

If you change the size of a window, the entire client area is set to invalid because of the CS\_HREDRAW and CS\_VREDRAW parameters. Therefore, the text is always displayed in the middle of the client area. If you decrease the size of the window so much that the text will no longer fit on one line in the rectangle, parts of the text will no longer be displayed. This occurs because the text would be written in areas protected by Windows. To make certain that the text is always displayed, replace the DT\_SINGLELINE flag with DT\_WORDBREAK.

In the second case the text is output in the same rectangle that was determined by the GetClientRect function. However, wcTextClass style is set to 0 in the WNDCLASS structure. A WM\_PAINT message is generated only when a window is enlarged and only the new part of the client area is clipped.

If you use the mouse to decrease the window so that its lower border is above the place where the text was previously located, the text no longer appears. The text also disappears if you enlarge the window slightly because the invalid area doesn't match the output area. With different arrangements the text may be visible again if the output is located in the rectangle that will be drawn.

The third case uses another rectangle for displaying the text. This is the invalid area that BeginPaint entered in the ps variable.

Instead of being called &rect, the fourth parameter of the DrawText function is now called &ps.rcPaint. Now the text is written in the middle of the clipping region.

Some attributes in the device context apply to text output. The SetTextColor function sets the color of the text. Use SetBkMode and SetBkColor to decide whether the space between the characters should be a different color (refer to the chapter on device context attributes).

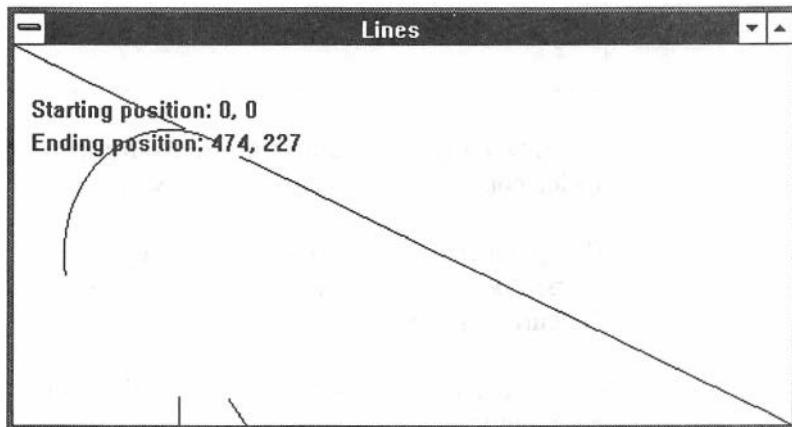
The background color and text color can also be based on the colors that are set by the Control Panel. If the system colors (in our example, window background color and text color) are changed and the

application then receives a WM\_PAINT message, the text and background will be displayed in the new colors.

```
SetTextColor( hDC, GetSysColor (COLOR_WINDOWTEXT) );
SetBkColor( hDC, GetSysColor (COLOR_WINDOW) );
```

The GetSysColor function retrieves the RGB value of the color that was specified using a color index. Each window part, whose color can be set by the Windows control panel, has a color index.

## Example of basic graphic functions



You can use basic graphic functions to draw lines as well as closed figures, such as rectangles and ellipses. These functions require coordinates in logical units so that the drawings are device-independent. Depending on which mapping mode is set in the device context, GDI converts the logical units to physical units (i.e., pixels).

Lines - Functions		
Function	Appearance	Description
<i>Arc</i>		Draws an arc
<i>LineDDA</i> (get from callback)		Calculates points on a line
<i>LineTo</i>		Draws a line
<i>MoveTo</i>		Changes current pen position
<i>Polyline</i>		Draws a series of lines

The LineTo function draws a line from the current pen position in the device context to the specified point, which is no longer enclosed.

This point represents the new current pen position. You could also reset the pen position with the MoveTo function and then prompt for it with GetCurrent Position.

LineTo is the only function that uses this position. LineTo and MoveTo are the only functions that change the position.

As the second parameter, the Polyline function is given a pointer to an array. This is a POINT array and contains points that should be connected to each other. The next parameter specifies the number of points.

Four points in x and y coordinates are passed to the Arc function. The first two points define a rectangle; the last two points define the start point and endpoint of the arc.

The last function, LineDDA, is more complicated. As the only output function, LineDDA doesn't require a handle to a device context.

DDA is an abbreviation for "Digital Differential Analyzer". LineDDA calculates all the points on a straight line between two defined points.

This function doesn't produce output itself. Instead it calls a callback function to produce output. The output isn't limited to only the calculated point. For example, it could also be an ellipse.

New messages	Brief description
WM_LBUTTONDOWN	Created when left mouse button is pressed
New functions	Brief description
Arc	Draws an arc
GetCurrentPosition	Determines the current pen position
LineTo	Draws a line
MoveTo	Changes the current pen position
Polyline	Draws a sequence of lines
New macros	Brief description
MAKEPOINT	Converts LONG variable to POINT
New structures	Brief description
POINT	Structure for x and y coordinates

## Source code: LINES.C

```
/** LINES.C ****
** Draws lines and figures using Arc, LineTo, MoveTo and PolyLine.      */
****

#include "windows.h"                                // Include windows.h header file
#include "string.h"                                 // Include string.h header file

BOOL LinesInit ( HANDLE );
long FAR PASCAL LinesWndProc( HWND, unsigned, WORD, LONG);

int PASCAL WinMain( hInstance, hPrevInstance, lpszCmdLine, cmdShow )
HANDLE hInstance, hPrevInstance;                  // Current & previous instance
LPSTR lpszCmdLine;                               // Long ptr to string after
                                                // program name during execution
int cmdShow;                                    // Specifies the application
                                                // window's appearance
{
    MSG msg;                                     // Message variable
    HWND hWnd;                                    // Window handle
```

## Output

---

```
if (!hPrevInstance)                                // Initialize first instance
{
    if (!LinesInit( hInstance )) // If initialization fails
        return FALSE;           // return FALSE
}

/** Specify appearance of application's main window *****/
hWnd = CreateWindow("Lines",
                    "Lines",                         // Window class name
                    WS_OVERLAPPEDWINDOW,             // Overlapped window style
                    CW_USEDEFAULT,                  // Default upper-left x pos.
                    0,                             // Upper-left y pos.
                    CW_USEDEFAULT,                  // Default initial x size
                    0,                             // y size
                    NULL,                          // No parent window
                    NULL,                          // Window menu used
                    hInstance,                     // Application instance
                    NULL);                        // No creation parameters

ShowWindow( hWnd, cmdShow );                      // Make window visible
UpdateWindow( hWnd );                            // Update window

while (GetMessage(&msg, NULL, 0, 0)) // Message reading
{
    TranslateMessage(&msg);          // Message translation
    DispatchMessage(&msg);          // Send message to Windows
}

return (int)msg.wParam;                           // Return wParam of last message
}

BOOL LinesInit( hInstance )                      // Line initialization
HANDLE hInstance;                               // Instance handle
{

/** Specify window class *****/
WNDCLASS      wcLinesClass;                   // Main window class

wcLinesClass.hCursor     = LoadCursor( NULL, IDC_ARROW );           // Mouse cursor
wcLinesClass.hIcon       = LoadIcon( NULL, IDI_APPLICATION );        // Default icon
wcLinesClass.lpszMenuName = NULL;                  // No menu
wcLinesClass.lpszClassName = "Lines";              // Window class
wcLinesClass.hbrBackground = (HBRUSH)GetStockObject( WHITE_BRUSH ); // White background
// White background
}
```

```

wcLinesClass.hInstance      = hInstance;                      // Instance
wcLinesClass.style         = CS_VREDRAW | CS_HREDRAW;        // Horizontal
                           // and vertical redraw of client area
wcLinesClass.lpfnWndProc   = LinesWndProc;                  // Window function
wcLinesClass.cbClsExtra    = 0 ;                            // No extra bytes
wcLinesClass.cbWndExtra    = 0 ;                            // No extra bytes

if (!RegisterClass( &wcLinesClass ) )           // Register window class
    return FALSE;                                // Return FALSE if registration fails
                                                   // If registration is successful,
return TRUE;                                 // return TRUE

}

/** LinesWndProc *****/
/** Main window function: All messages are sent to this window */
/** *****/

long FAR PASCAL LinesWndProc( hWnd, message, wParam, lParam )
HWND      hWnd;                                         // Window handle
unsigned   message;                                     // Message type
WORD       wParam;                                      // Message-dependent 16 bit value
LONG       lParam;                                     // Message-dependent 32 bit value
{

    HDC          hdc;                                    // Handle to device context
    PAINTSTRUCT ps;                                   // Paint structure declaration
    RECT         rect;                                  // RECT rectangle structure
    static POINT pt;                                  // POINT point structure
    static POINT ptarray[4] = { 100,210, 100,250, 150,240, 130,210 };
    LONG         lPoint;
    char         chText[50];

    switch (message)                                // Process messages
    {
        case WM_LBUTTONDOWN:                         // Left button clicked? Move
            pt = MAKEPOINT (lParam); // starting position to
            InvalidateRect(hWnd, NULL, TRUE); // new location
            break;                                // End of this message process

        case WM_PAINT:                             // Client area redraw
            hdc = BeginPaint (hWnd, &ps); // Begin WM_PAINT
                                               // Set, display starting position
            MoveTo(hdc, pt.x, pt.y);
            wsprintf(chText, "Starting position: %d, %d", pt.x, pt.y);
            TextOut(hdc, 10, 30, chText, strlen(chText));
                                               // Set, display ending position
            GetClientRect(hWnd, &rect); // Get client rectangle
    }
}

```

## *Output*

---

```
        LineTo(hDC, rect.right, rect.bottom); // Draw line
        lPoint = GetCurrentPosition(hDC);
        pt = MAKEPOINT(lPoint);
        wsprintf(chText, "Ending position: %d, %d", pt.x, pt.y);
        TextOut(hDC, 10, 50, chText, strlen(chText));

        pt.x = 0;                      // Starting position default
        pt.y = 0;                      // Draw multiple lines ...
        Polyline(hDC, (LPPOINT)&ptarray, (int)4);
        Arc(hDC, 30,50,160,200, 140,20, 20,140); // and arc

        EndPaint (hWnd, &ps); // End WM_PAINT process
        break;                  // End of this message process

    case WM_DESTROY:           // Destroy window
        PostQuitMessage(0);
        break;

    default:                  // Send other messages to
        // default window function
        return (DefWindowProc( hWnd, message, wParam, lParam ));
        break;
    }
    return(0L);
}
```

## **Module definition file: LINES.DEF**

NAME	Lines
DESCRIPTION	'Output example'
EXETYPE	WINDOWS
STUB	'WINSTUB.EXE'
CODE	PRELOAD MOVEABLE
DATA	PRELOAD MOVEABLE MULTIPLE
HEAPSIZE	4096
STACKSIZE	4096
EXPORTS	LinesWndProc @1

---

To compile and link this application, use the COMPILE.BAT batch file described earlier in this book. You can create this file using the DOS COPY CON command or any text editor or word processor that generates ASCII files. Here's the listing again:

```
cl -c -Gw -Zp %1  
link /align:16 %1,%1.exe,,libw+slibcew,%1.def  
rc %1.exe
```

Save this file to a directory contained in your path. Then when you are in the directory containing your source and definition files, type the following and press **Enter**:

```
compile lines
```

The batch file performs all the tasks needed.

## How LINES.EXE works

The application uses the LineTo function to draw a line from the current position to the lower-right corner of the client area, which is calculated with the help of the GetClientRect function. The default value of the pen position is (0,0) and is recorded in the pt variable.

Therefore, a diagonal line is drawn from the upper-left corner to the lower-right corner for the first output and all other output that isn't blocked by pressing the left mouse button.

The endpoint, which is now also the current pen position, is calculated with the GetCursorPosition function and output by TextOut.

Pressing the left mouse button transfers the point where the cursor is located to the pt variable and generates a WM\_PAINT message. The MAKEPOINT macro obtains this point from lParam. While the WM\_PAINT message is being processed, the MoveTo function sets the current position to this point and outputs it with TextOut.

Finally the start point pt is erased because when the device context is released, the current pen position is automatically set back.

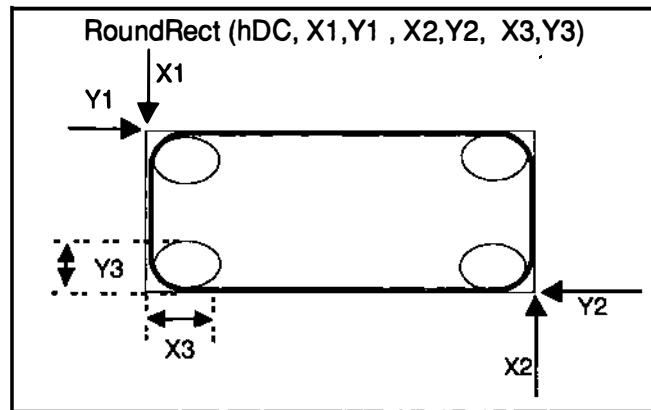
Three connected lines and an arc with fixed point values are drawn. If the window is smaller than the specified coordinates, the figures are only partially displayed.

Ellipse and Polygon Functions		
Function	Appearance	Description
Chord		Draws a secant
Ellipse		Draws an ellipse
Pie		Draws a pie
Polygon		Draws a polygon
Polypolygon		Draws a series of polygons
Rectangle		Draws a rectangle
RoundRect		Draws a round rectangle

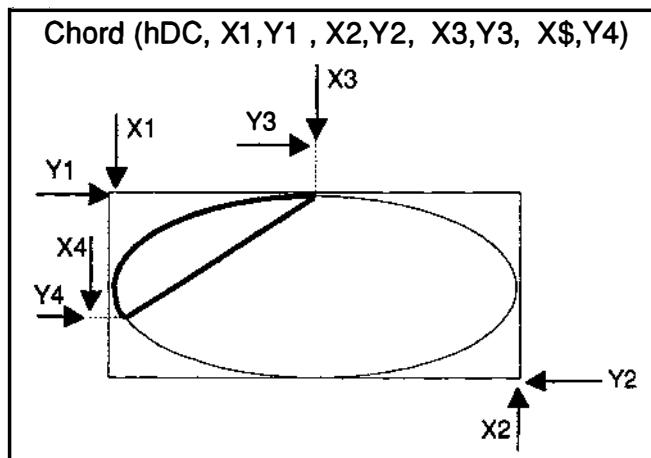
Except for the two polygon functions, the structure of the other functions is the same. The first parameter is usually the handle to a device context. The second and third parameters determine the upper-left corner.

The fourth and fifth parameters determine the lower-right corner of a rectangle, which is either displayed or used as the basis for the figure that will be drawn.

So the RoundRect function (see the following illustration) also needs two parameters that define the height and width of the ellipses, which are responsible for the rounded corners.



Two additional points, which are specified in x and y coordinates, inform the Chord function of the location where the line interacts with the arc, which is part of an ellipse. These two points don't have to be in the invisible rectangle. The Pie function and the Arc function receive the same transfer values.



You can use the two polygon functions to draw any figure. The Polygon function has the same parameters as the Polyline function except that it also connects the start point with the endpoint.

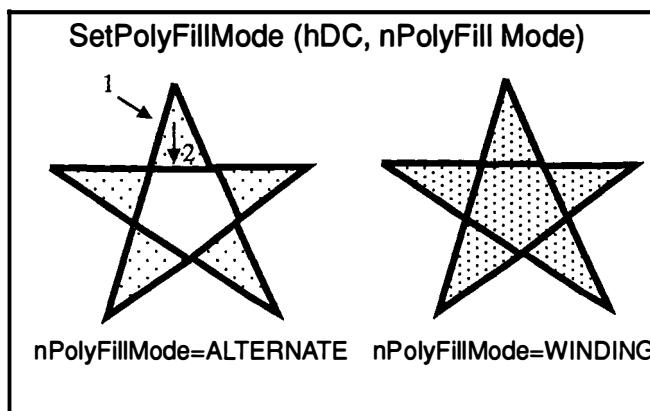
The PolyPolygon function creates several closed polygons that are partially filled with the current brush on the basis of the ALTERNATE fill mode. The current setting of the fill mode doesn't matter. Although the polygons can overlap, this isn't mandatory.

## *Output*

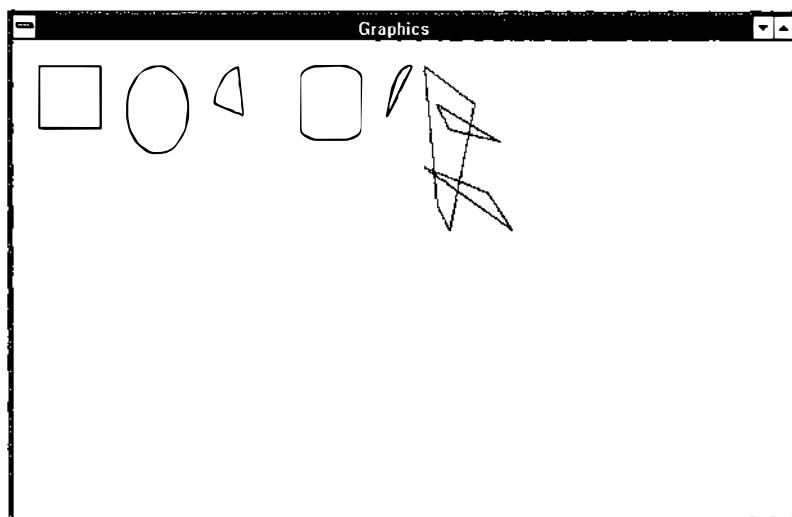
---

The SetPolyFillMode function sets the fill mode for polygons. This function has two settings: ALTERNATE and WINDING. Depending on the mode that is set, the current brush either fills all or only certain areas of a closed polygon.

If you want to fill all of the areas, pass the WINDING keyword to the function as the second parameter. ALTERNATE is the more complicated setting. Only the restricted areas that can be reached from the outside using an odd number of lines are filled. The brush cannot cross over intersections.



## **Example of ellipse and polygon output**



The following application demonstrates some of the options of the basic graphic functions of the graphic device interface (GDI). Various figures are drawn while the WM\_PAINT message is being processed: a rectangle, an ellipse, a sector of a circle, a secant, and several polygons.

As you can see, the function parameters for the first five figures are very similar.

New functions	Brief description
Chord	Draws a secant
Ellipse	Draws an ellipse
Pie	Draws a pie
PolyPolygon	Draws several polygons
Rectangle	Draws a rectangle
RoundRect	Draws a rectangle with rounded corners

## Source code: GRAPHICS.C

```
/** GRAPHICS.C ****
/** Draws a series of graphics using Windows commands.      */
/****

#include "windows.h"                                // Include windows.h header file

long FAR PASCAL GraphicWndProc ( HWND, unsigned, WORD, LONG );
BOOL GraphicInit ( HANDLE );

int PASCAL WinMain ( hInstance, hPrevInstance, lpszCmdLine, cmdShow )
HANDLE hInstance, hPrevInstance;                      // Current & previous instance
LPSTR lpszCmdLine;                                  // Long ptr to string after
                                                    // program name during execution
int cmdShow;                                         // Specifies the application
                                                    // window's appearance
{
    MSG msg;                                         // Message variable
    HWND hWnd;                                       // Window handle

    if (!hPrevInstance)                             // Initialize first instance
    {
        if (!GraphicInit( hInstance )) // If initialization fails
            return FALSE;                         // return FALSE
    }
}
```

## Output

---

```
/** Specify appearance of application's main window *****/
hWnd = CreateWindow("Graphics",           // Window class name
                    "Graphics",          // Window caption
                    WS_OVERLAPPEDWINDOW, // Overlapped window style
                    CW_USEDEFAULT,       // Default upper-left x pos.
                    0,                  // Upper-left y pos.
                    CW_USEDEFAULT,       // Default initial x size
                    0,                  // y size
                    NULL,               // No parent window
                    NULL,               // Window menu used
                    hInstance,           // Application instance
                    NULL);              // No creation parameters

ShowWindow( hWnd, cmdShow );            // Make window visible
UpdateWindow( hWnd );                 // Update window

while (GetMessage(&msg, NULL, 0, 0)) // Message reading
{
    TranslateMessage(&msg);         // Message translation
    DispatchMessage(&msg);         // Send message to Windows
}

return msg.wParam;                     // Return wParam of last message
}

BOOL GraphicInit( HANDLE hInstance )      // Timer initialization,
                                            // instance handle
{

/** Specify window class *****/
WNDCLASS      wcGraphicClass;           // Main window class

wcGraphicClass.style      = CS_VREDRAW | CS_HREDRAW;
                           // Horizontal and vertical
                           //   redraw of client area
wcGraphicClass.hCursor    = LoadCursor( NULL, IDC_ARROW );
                           // Mouse cursor
wcGraphicClass.hIcon      = LoadIcon( NULL, IDI_APPLICATION );
                           // Default icon
wcGraphicClass.lpszMenuName = NULL;        // No menu
wcGraphicClass.lpszClassName = "Graphics"; // Window class
wcGraphicClass.hbrBackground = GetStockObject( WHITE_BRUSH );
                           // White background
wcGraphicClass.hInstance   = hInstance;      // Instance
wcGraphicClass.lpfnWndProc = GraphicWndProc; // Window function
```

```
wcGraphicClass.cbClsExtra      = 0;                      // No extra bytes
wcGraphicClass.cbWndExtra     = 0;                      // No extra bytes

if (!RegisterClass( (LPWNDCLASS) &wcGraphicClass ) )
    // Register window class
    return FALSE;
return TRUE;                                         // If registration is successful,
}                                                       // return TRUE

/** GraphicWndProc *****/
/** Main window function: All messages are sent to this window */
/** *****/

long FAR PASCAL GraphicWndProc( hWnd, message, wParam, lParam )
HWND      hWnd;                                     // Window handle
unsigned   message;                                // Message type
WORD       wParam;                                 // Message-dependent 16 bit value
LONG      lParam;                                // Message-dependent 32 bit value
{

PAINTSTRUCT ps;                                    // Paint structure declaration
HDC        hDC;                                    // Device context handle
static POINT ptarray[] = {330,20, 340,130, 350,150, 370,50, 330,20,
                          340,50, 350,70, 390,80, 340,50,
                          330,100, 400,150, 380,120, 330,100 };
static int iarray[] = {5, 4, 4 };

switch (message)                                     // Process messages
{
    case WM_PAINT:                                // Client area redraw
        hDC = BeginPaint( hWnd, &ps ); // Begin WM_PAINT

/** Draw figures ****/

        Rectangle (hDC, 20,20, 70, 70);
        Ellipse   (hDC, 90,20, 140, 90);
        RoundRect (hDC, 230,20, 280, 80, 30, 20);
        Pie       (hDC, 160,20, 210,100, 180,20, 160,50);
        Chord     (hDC, 300,20, 340,120, 320,10, 300,60);
        PolyPolygon(hDC, (LPPOINT)&ptarray, (LPINT)iarray,3);

        EndPaint( hWnd, &ps ); // End WM_PAINT
        break;                               // End of this message process

    case WM_DESTROY:                            // Destroy window
        PostQuitMessage(0);
        break;
}
```

## Output

---

```
        default:          // Send other messages to
                           // default window function
            return (DefWindowProc( hWnd, message, wParam, lParam ));
        break;           // End of this message process
    }
return(0L);
}
```

## Module definition file: GRAPHICS.DEF

```
NAME      Graphics

DESCRIPTION 'Draws ellipses, polygons, etc.'

EXETYPE   WINDOWS

STUB      'WINSTUB.EXE'

CODE      PRELOAD MOVEABLE
DATA      PRELOAD MOVEABLE MULTIPLE

HEAPSIZE  4096
STACKSIZE 4096

EXPORTS   GraphicWndProc    @1
```

To compile and link this application, use the COMPILE.BAT batch file described earlier in this book. You can create this file using the DOS COPY CON command or any text editor or word processor that generates ASCII files. Here's the listing again:

```
cl -c -Gw -Zp %1
link /align:16 %1,%1.exe,,libw+slibcew,%1.def
rc %1.exe
```

Save this file to a directory containing your path or one containing the source and definition files. Type the following and press **Enter**:

```
compile graphics
```

## How GRAPHICS.EXE works

Similar to the Polyline function, the PolyPolygon function is passed a pointer to an array of the POINT structure as its second parameter. All

the points are specified here, regardless of the polygon to which they belong.

The next parameter includes a pointer to an array of integer values that determine the number of points a polygon has. The last parameter contains the total number of these integer values.

Unlike the Polygon function, these polygons aren't automatically closed (i.e., the first point of a polygon must be named twice). Although the polygon fill mode ALTERNATE occurs, the results aren't visible because both the default mode and the background are white.

## Drawing tools

It's possible to change some of the GDI painting tools to vary the output. The pen, brush, and font are the most important drawing tools.

You don't have to reset these tools each time you output something because their current values are attributes in the device context. So these values are automatically used when the device context receives a handle.

When you work with different attributes, you should follow a specific procedure:

- Obtain device context
- Get drawing tool
- Select tool in the device context
- Output functions
- Select old tool in the device context. Then you can delete the first tool
- Release device context

You can either work with the default settings or create your own tools. All the default tools are already in the Windows system. To set a different default attribute, you need the GetStockObject function, which provides the handle to a default tool. This function's only parameter is a constant that is passed to it. The constant sets the desired default tool (e.g., GetStockObject(WHITE\_BRUSH)).

All self-defined tools have functions that begin with the word "Create". These functions generate new tools and receive a handle. These handles have the following data types: HPEN, HBRUSH, HFONT.

## Using the tools

Since you don't need a device context to obtain a handle to a tool, the procedure for obtaining the tool and the device context could be different than previously described.

First tell the device context that it must reset an attribute. The SelectObject function does this by obtaining a handle to the device context and a handle to the new tool. This function retrieves the handle of the tool that was current until this point.

Since the device context should have the same attribute at the end that it did in the beginning, you must retain this handle. After the SelectObject function has been executed, all output considers the new tool.

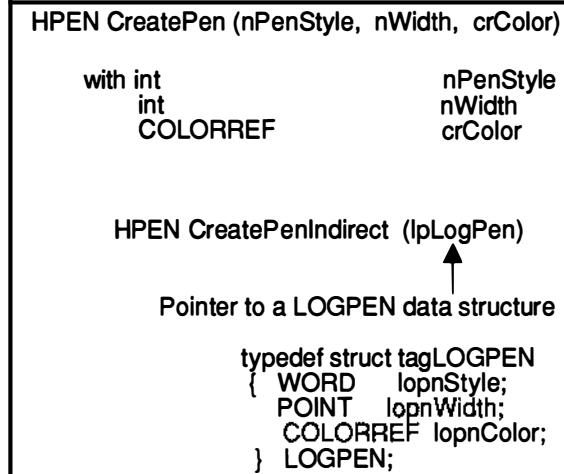
When the selected brush, pen, or font is no longer being used, the same function can be used to enter the old tool in the device context again. DeleteObject deletes all tools that aren't default tools because only a limited number of handles exist. The device context is released at the end of output.

## Pen

The pen tool draws all types of lines, from simple lines (LineTo) to figures such as a secant (Chord). The pen determines three qualities: width, color, and style. Windows includes the following default pens:

`BLACK_PEN`  
`WHITE_PEN`  
`NULL_PEN`

The `NONE_PEN` can draw a figure, such as the ellipse, without a border. If you want the lines to be wider or dotted, you must define your own pen with either `CreatePen` or `CreatePenIndirect`. The only difference between these functions and standard functions is that you pass parameters.



The line width is specified in logical units. In MM\_TEXT mapping mode this means that one pixel represents a logical unit. In the LOGOPEN structure, the width of lopnWidth is of the POINT type. However, the y coordinates aren't used.

You can choose from seven different styles:

Different pen styles		
Style	Value	Appearance
PS_SOLID	0	—————
PS_DASH	1	- - - - -
PS_DOT	2	.....
PS_DASHDOT	3	- - - -
PS_DASHDOTDOT	4	— - - -
PS_NULL	5	
PS_INSIDEFRAME	6	

If you set the PS\_INSIDEFRAME style and the width is greater than 1, the line will be drawn within the frame for all basic graphics functions except polygons and polylines.

With these two functions and a pen width that is equal to or less than 1, the PS\_INSIDEFRAME style is identical to PS\_SOLID. You can use the styles 0, 5, or 6 to combine wider lines.

The color is of the COLORREF data type, behind which follows a long integer. A COLORREF variable can name the color in three variants. The color can be specified as an RGB value, an index on a logical palette entry, or a palette relative RGB value.

The last two options will be discussed in detail in a separate chapter. The RGB value can either be written in hexadecimal notation or with the RGB macro.

The hexadecimal notation is as follows: 0x00bbggrr, with 'b' for blue, 'g' for green and 'r' for red. The fields of each primary color define the color intensity. The value 255 (decimal) or FF (hexadecimal) indicates high intensity. The byte with the highest value must always have the value 00.

Hex value	Color
0x000000FF	Pure red
0x0000FF00	Pure red
0x00FF0000	Pure blue
0x00000000	Black
0x00FFFFFF	White

The RGB macro offers another method of color selection. The three primary colors are represented by a byte that can accept values between 0 and 255. The sequences of colors to be specified is red, green, and blue. Setting all the values to 0 selects black.

If the GDI receives an RGB value as a function parameter, it passes the RGB color value directly to the output device driver. The output device driver then selects the color that's closest to the desired color. Since many color output devices can only display a certain number of colors, the current color is simulated by mixing pixels and displaying colors that can appear in different ways.

If the device is only capable of monochrome display, the driver selects either white, black, or various shades of grey. If, in such a case, the sum of the RGB values is larger than 765, the color is white.

## **Brush**

A brush is a bitmap with dimensions of 8 x 8 pixels. This bitmap is repeated horizontally and vertically until the interior of a closed area is

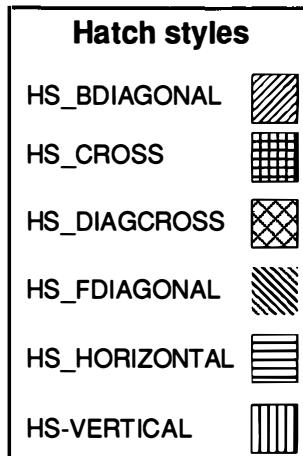
filled. This area was created by one of the basic ellipse and polygon functions.

Windows contains seven default brushes. Use GetStockObject to obtain the handles of the brushes.

```
BLACK_BRUSH  
DKGRAY_BRUSH  
GRAY_BRUSH  
HOLLOW_BRUSH  
LTGRAY_BRUSH  
NULL_BRUSH  
WHITE_BRUSH
```

You can also create your own brushes. Like the pens, a function (CreateBrushIndirect) is passed an initialized structure (LOGBRUSH). Depending on the kind of brush, you can use one of four functions to specify the parameters directly.

There are solid brushes, hatched brushes, and brushes containing a pattern that's saved in a bitmap. The bitmap can also be a DIB (Device Independent Bitmap). There are six different kinds of hatches.



```
HBRUSH CreateSolidBrush (crColor)  
with COLORREF crColor: color  
  
HBRUSH CreateHatchBrush ( nIndex, crColor)  
with int nIndex: Hatch style
```

## *Output*

---

```
COLORREF           crColor: color

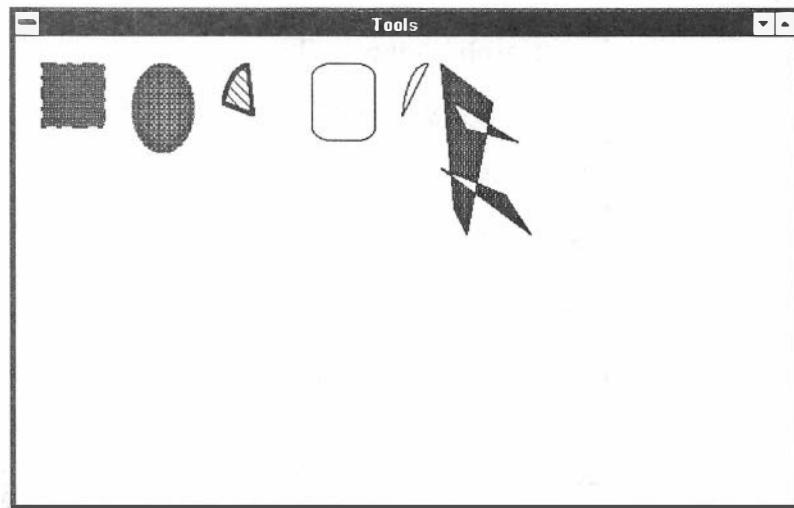
HBRUSH CreatePatternBrush (hBitmap)
with   HBITMAP        hBitmap: Handle to a bitmap

HBRUSH CreateDIBPatternBrush (hPackedDIB, wUsage)
with   GLOBALHANDLE    hPackedDIB: Handle to global memory object with packed DIB
      WORD            wUsage: how the color table should be interpreted
```

Unlike the LOGBRUSH structure, these functions never need all three parameters. Instead these functions only need one or two values. So it's easy to supply these values especially since the second and third field of the LOGBRUSH structure depends on the contents of the first field.

```
typedef struct tagLOGBRUSH
{
    WORD      lbStyle;
    COLORREF  LBColor;
    short int  lbHatch;
} LOGBRUSH;
```

## **Example of Pen and Brush tools**



To experiment with different pens and brushes, this application is an expansion of the GRAPHIC application listed previously. The TOOLS.C source code also includes the ALTERNATE polygon fill mode.

---

New functions	Brief description
CreateHatchBrush	Creates a logical brush with a hatch pattern
CreatePen	Creates a logical pen
CreateSolidBrush	Creates a solid logical brush
DeleteObject	Deletes created tool
GetStockObject	Obtains the handle of a default tool
SelectObject	Places the tool in the device context as the current tool

## Source code: TOOLS.C

```
/** TOOLS.C ****
/** Draws a series of graphics using different line and fill tools      */
/***** */

#include "windows.h"                                // Include windows.h header file

long FAR PASCAL ToolsWndProc ( HWND, unsigned, WORD, LONG);
BOOL ToolsInit ( HANDLE );

int PASCAL WinMain ( hInstance, hPrevInstance, lpszCmdLine, cmdShow )
HANDLE hInstance, hPrevInstance;                  // Current & previous instance
LPSTR lpszCmdLine;                            // Long ptr to string after
                                                // program name during execution
int cmdShow;                                  // Specifies the application
                                                // window's appearance

{
    MSG msg;                                    // Message variable
    HWND hWnd;                                 // Window handle

    if (!hPrevInstance)                         // Initialize first instance
    {
        if (!ToolsInit( hInstance )) // If initialization fails
            return FALSE;                      // return FALSE
    }
}

/** Specify appearance of application's main window ****

hWnd = CreateWindow("Tools",                // Window class name
                    "Tools",                 // Window caption
                    WS_OVERLAPPEDWINDOW, // Overlapped window style
                    CW_USEDEFAULT,       // Default upper-left x pos.
                    0,                     // Upper-left y pos.
                    CW_USEDEFAULT,       // Default initial x size
                    0,                     // y size
                    NULL,                 // No parent window

```

## Output

---

```
        NULL,                                // Window menu used
        hInstance,                            // Application instance
        NULL);                               // No creation parameters

ShowWindow( hWnd, cmdShow );           // Make window visible
UpdateWindow( hWnd );                // Update window

while (GetMessage(&msg, NULL, 0, 0)) // Message reading
{
    TranslateMessage(&msg);            // Message translation
    DispatchMessage(&msg);           // Send message to Windows
}

return msg.wParam;                   // Return wParam of last message
}

BOOL ToolsInit( HANDLE hInstance )      // Tools initialization,
                                         // instance handle

/** Specify window class *****/
{

    WNDCLASS      wcToolsClass;          // Main window class

    wcToolsClass.style      = CS_VREDRAW | CS_HREDRAW; // Horizontal and
                                                // vertical redraw of client area
    wcToolsClass.hCursor    = LoadCursor( NULL, IDC_ARROW );
                                         // Mouse cursor
    wcToolsClass.hIcon      = LoadIcon( NULL, IDI_APPLICATION );
                                         // Default icon
    wcToolsClass.lpszMenuName = NULL;       // No menu
    wcToolsClass.lpszClassName = "Tools";   // Window class
    wcToolsClass.hbrBackground = GetStockObject( WHITE_BRUSH );
                                         // White background
    wcToolsClass.hInstance   = hInstance;    // Instance
    wcToolsClass.lpfnWndProc = ToolsWndProc; // Window function
    wcToolsClass.cbClsExtra = 0;           // No extra bytes
    wcToolsClass.cbWndExtra = 0;           // No extra bytes

    if (!RegisterClass( (LPWNDCLASS) &wcToolsClass ) )           // Register
                                                               // window class
        return FALSE;           // Return FALSE if registration fails
                                         // If registration is successful,
                                         // Return TRUE
    return TRUE;
}
```

```
/** ToolsWndProc ****
** Main window function: All messages are sent to this window      */
*****  
  
long FAR PASCAL ToolsWndProc( hWnd, message, wParam, lParam )
HWND      hWnd;                                // Window handle
unsigned   message;                            // Message type
WORD       wParam;                             // Message-dependent 16 bit value
LONG       lParam;                            // Message-dependent 32 bit value
{  
  
    PAINTSTRUCT ps;                         // Paint structure declaration
    HDC        hDC;                           // Device context handle
    HPEN      hPenNew, hPenOld;                // Pens
    HBRUSH    hBrushNew, hBrushOld;              // Brushes  
  
    static POINT     ptarray[] = {330,20, 340,130, 350,150, 370,50, 330,20,
                                    340,50, 350,70, 390,80, 340,50,
                                    330,100, 400,150, 380,120, 330,100 };
    static int      iarray[] = {5, 4, 4 };  
  
    switch (message)                      // Process messages
    {
        case WM_PAINT:                   // Client area redraw
            hDC = BeginPaint( hWnd, &ps ); // Begin WM_PAINT  
  
    /** Graphic commands ****
***** Rectangle *****  
  
        hPenNew = CreatePen( PS_DASHDOT, 1, RGB(0,0,255));
        hPenOld = SelectObject( hDC, hPenNew);  
  
        hBrushNew = GetStockObject(GRAY_BRUSH);
        hBrushOld = SelectObject( hDC, hBrushNew);  
  
        Rectangle (hDC, 20,20, 70, 70);  
  
        SelectObject (hDC, hPenOld);
        DeleteObject (hPenNew);  
  
        SelectObject (hDC, hBrushOld);  
  
    ***** Ellipse *****  
  
        hPenNew = GetStockObject(NULL_PEN);
        hPenOld = SelectObject (hDC, hPenNew);
```

## Output

---

```
hBrushNew = CreateSolidBrush (RGB(255,0,0));
hBrushOld = SelectObject( hDC, hBrushNew);

Ellipse    (hDC,   90,20, 140, 90);

SelectObject (hDC, hPenOld);

DeleteObject( SelectObject (hDC, hBrushOld));

RoundRect (hDC, 230,20, 280, 80, 30, 20);

/***************************************** Pie *****/
hPenNew = CreatePen (PS_SOLID, 3, 0x00000000);
hPenOld = SelectObject (hDC, hPenNew);

hBrushNew = CreateHatchBrush (HS_FDIAGONAL, 0x000000FF);
hBrushOld = SelectObject( hDC, hBrushNew);

Pie        (hDC, 160,20, 210,100, 180,20, 160,50);

DeleteObject( SelectObject (hDC, hPenOld));
DeleteObject( SelectObject (hDC, hBrushOld));

Chord      (hDC, 300,20, 340,120, 320,10, 300,60);

/***************************************** Polygon fill mode *****/
hBrushNew = GetStockObject (DKGRAY_BRUSH);
hBrushOld = SelectObject (hDC, hBrushNew);

PolyPolygon(hDC, (LPPOINT)&ptarray, (LPINT)iarray,3);

SelectObject (hDC, hBrushOld);

EndPaint( hWnd, &ps ); // End WM_PAINT
break;                // End of this message process

case WM_DESTROY:           // Destroy window
    PostQuitMessage(0);
    break;                  // End of this message process

default:                  // Send other messages to
                        // default window function
return (DefWindowProc( hWnd, message, wParam, lParam ));
break;                  // End of this message process
```

```
}
```

```
return(0L);
```

```
}
```

## Module definition file: TOOLS.DEF

```
NAME          Tools
```

```
DESCRIPTION 'Example of drawing tools'
```

```
EXETYPE      WINDOWS
```

```
STUB          'WINSTUB.EXE'
```

```
CODE          PRELOAD MOVEABLE
```

```
DATA          PRELOAD MOVEABLE MULTIPLE
```

```
HEAPSIZE     4096
```

```
STACKSIZE    4096
```

```
EXPORTS       ToolsWndProc    @1
```

To compile and link this application, use the COMPILE.BAT batch file described earlier in this book. You can create this file using the DOS COPY CON command or any text editor or word processor that generates ASCII files. Here's the listing again:

```
cl -c -Gw -Zp %1
link /align:16 %1,%1.exe,,libw+slibcew,%1.def
rc %1.exe
```

Save this file to a directory contained in your path. Then when you are in the directory containing your source and definition files, type the following and press **Enter**:

```
compile tools
```

The batch file performs all the tasks needed.

## How **TOOLS.EXE** works

This example demonstrates the procedure for using tools. First a handle was obtained so that the new tool could be selected in the device context. A pen was defined and the predefined **GRAY\_BRUSH** was used for the rectangle.

After drawing the rectangle, the old tool must be reselected using the **SelectObject**. Then only the created tools are deleted with **DeleteObject**. Other tools are selected for the ellipse and the pie.

To demonstrate the **ALTERNATE** polygon fill mode, which always appears with the **PolyPolygon** function, a brush other than the **WHITE\_BRUSH** must be used. This is necessary because the background of the entire client area is also white (refer to the section on basic graphic functions).

**Drawing mode:** Use this mode to specify which logical (Boolean) operation runs in reference to the color between the pixels of the pen or brush and the pixels of the target screen. This logical operation is also called a Raster Operation (ROP).

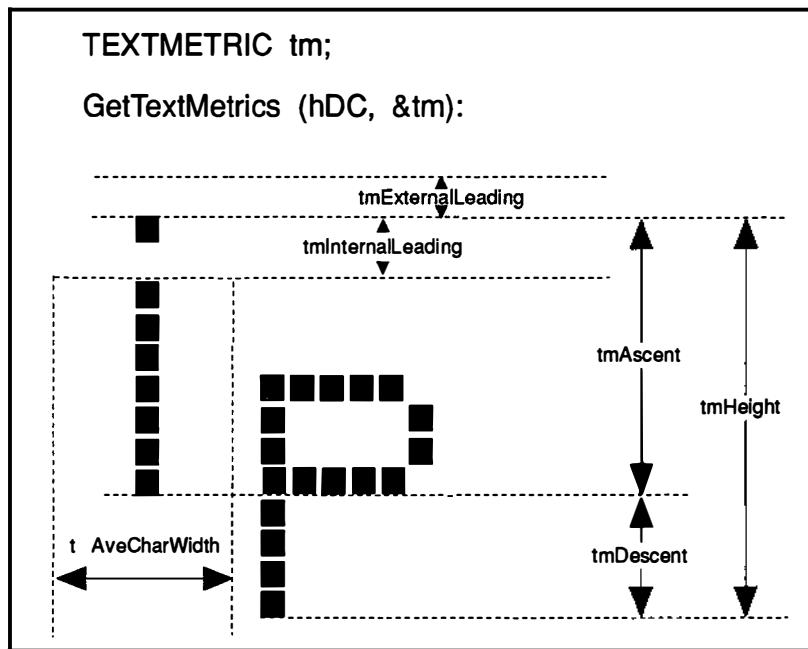
Since, in this case, there are two input parameters (the text color and the background color), this mode is also called Binary Raster Operation and is specified by the **ROP2** Code. In the device context, **R2\_COPYPEN** is predefined as the drawing mode. This means that the color of the pen is always accepted, regardless of the color of the background.

Sixteen different combinations can be selected through a name. These combinations occur in a monochrome system providing four options (white pen on white background, white pen on black background, black pen on white background, and black pen on black background), which can be combined with each other sixteen different ways.

The SetROP2 function changes the ROP2 code. For example, the R2\_BLACK ROP2 code draws only black lines.

## Font

Like the current pen and brush, the current font is in the device context. Before using the TextOut function to fill the client area with several lines of text, you must determine the current client area size and font size.



The GetTextMetrics function determines the character dimensions from the device context and places them in a variable of the TEXTMETRIC structure (refer to the illustration above). This structure contains 20 fields.

All sizes are given in logical units so that they depend on the current mapping mode. In addition to the information about the font family, the size of a letter is also indicated.

## *Output*

---

The `tmExternalLeading` parameter, which is defined by the font designer, specifies the space between the single rows of text.

Since many programs use the system font, the height and width of a letter can be determined with the `WM_CREATE` message:

```
static short xChar, yChar;  
WM_CREATE:  
    hdc = GetDC (hWnd);  
    GetTextMetrics (hdc, &tm);  
    xChar = tm.tmAveCharWidth;  
    yChar = tm.tmHeight + tm.tmExternalLeading;  
    ReleaseDC (hWnd, hdc);  
    break;
```

The size of the client area is also important. This size can be determined with either the `GetClientRect` function or the `WM_SIZE` message. A `WM_SIZE` message occurs each time the window is enlarged or reduced. The `lParam` parameter specifies the new height and width of the client area.

```
static short xClient, yClient;  
WM_SIZE:  
    xClient = LOWORD(lParam);  
    yClient = HIWORD(lParam);  
    break;
```

With these values you can determine the number of possible lines (`yClient/yChar`) and the number of possible letters in a line (`xClient/xChar`).

Windows provides five default fonts. Use `GetStockObject` to obtain their handles and `SelectObject` to select them in the device context.

<b>Default font</b>	<b>Description</b>
<code>SYSTEM_FONT</code>	Denotes the system font, which is a fixed pitch font with the ANSI character set. This font is always available.
<code>OEM_FIXED_FONT</code>	Denotes a fixed pitch font with an OEM character set, which can vary from system to system.
<code>ANSI_FIXED_FONT</code>	Denotes a fixed pitch font with an ANSI character set. A Courier font is used if available. Its letters are smaller than the system font.

**ANSI\_VAR\_FONT**

Denotes a font with variable letter width based on the ANSI character set. If available, a Helvetica or Times Roman font is used.

**DEVICE\_DEFAULT\_FONT**

Denotes a font preferred by the device. If no font is preferred, either a Courier font or the system font is used.

In the GDI, each letter is surrounded by a rectangle called the character cell. This rectangle consists of a fixed number of lines and columns.

The term fixed pitch refers to a font whose character cells are the same size. However, the sizes of these cells vary in a proportional font.

If the default fonts aren't sufficient, you can use either CreateFont or CreateFontIndirect to create your own logical fonts.

The LOGFONT structure is slightly larger than the LOGPEN and LOGBRUSH structures.

## LOGFONT structure:

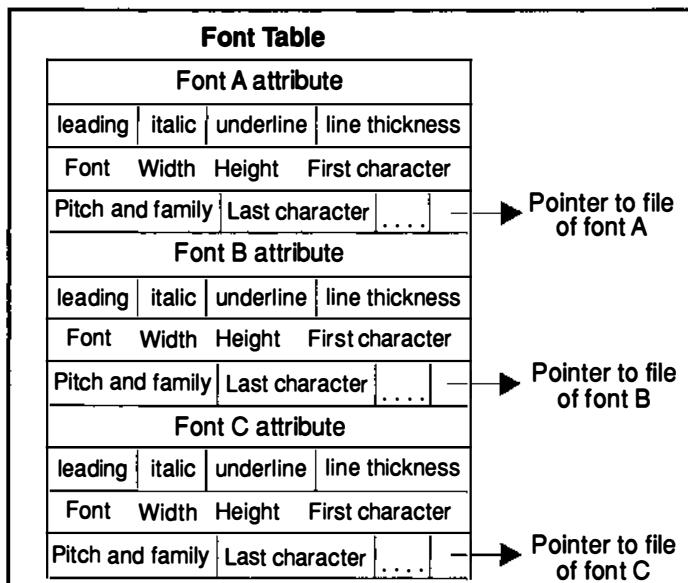
Field	Data type	Brief description
IfHeight	short int	Height of font in logical units
IfWidth	short int	Average width of a character
IfEscapement	short int	Angle of each line of text in relation to the lower edge of the paper
IfOrientation	short int	Angle of every base line of a character in relation to the lower edge of the paper
IfWeight	short int	Font weight: 400 Normal 700 Bold
IfItalic	BYTE	italic, if set to TRUE
IfUnderline	BYTE	underlined, if set to TRUE
IfStrikeOut	BYTE	strikeout if set to TRUE
IfCharSet	BYTE	Character set: ANSI_CHARSET: 0 OEM_CHARSET: 255
IfOutPrecision	BYTE	Precision of output: OUT_CHARACTER_PRECIS: 2 OUT_DEFAULT_PRECIS: 0 OUT_STRING_PRECIS: 1 OUT_STROKE_PRECIS: 3
IfClipPrecision	BYTE	Precision of clippings: CLIP_CHARACTER_PRECIS: 1 CLIP_DEFAULT_PRECIS: 0 CLIP_STROKE_PRECIS: 2
IfQuality	BYTE	Output quality: DEFAULT_QUALITY: 0 DRAFT_QUALITY: 1 PROOF_QUALITY: 2
IfPitchAndFamily	BYTE	Pitch and font family, linked with OR Pitch: DEFAULT_PITCH: 0 FIXED_PITCH: 1 VARIABLE_PITCH: 2  Family: FF_DECORATIVE: (5<<4) FF_DONTCARE: (0<<4) FF_MODERN: (3<<4) FF_ROMAN: (1<<4) FF_SCRIPT: (4<<4) FF_SWISS: (2<<4)
IfFaceName[]	BYTE	Typeface name of font

The LOGFONT structure is very similar to the TEXTMETRIC structure, which is responsible for the current physical font.

When selecting a logical font in the device context, the GDI selects a physical, existing font that best matches the requirements of the logical

font. The font could match the logical font exactly or be completely different.

The GDI contains a font table that's created when Windows is installed and can be expanded using the control panel. Each entry in this table describes a physical font and its attributes. Each entry also has a pointer that refers to the corresponding font resource. The font resource is a file with the .FON extension. These files contain one or more sizes of a specific font.



Occasionally it's possible to use the desired font even if an identical physical font isn't available. For example, often the GDI can adjust the size of the letters. Suppose that you want a physical font that has a letter height of ten logical units. However, only a physical font with a letter height of five logical units is available.

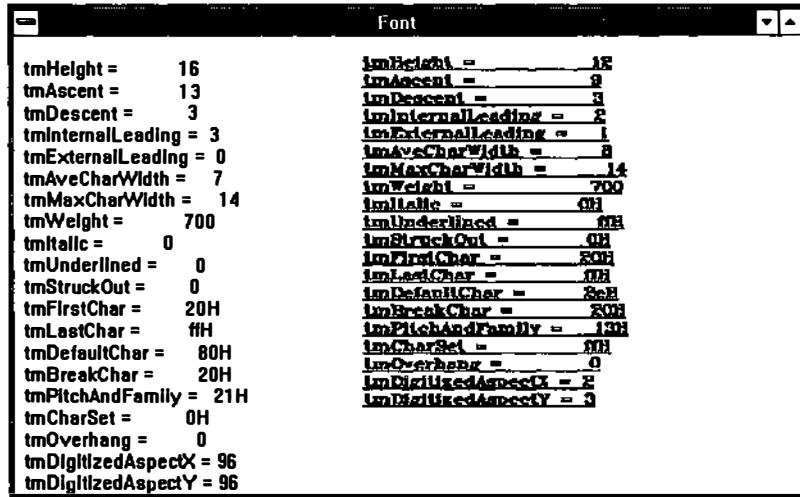
The `SelectObject` function can artificially produce the desired font by doubling the height. Windows frequently uses this process with the italic, bold, underline, and strikeout attributes. Otherwise, a separate font would be needed for each attribute.

When comparing the logical fonts with the fonts in the font table, the GDI gives penalty points to the physical fonts that don't have the desired

characteristics. These characteristics are ranked in order of importance. If the character set doesn't match, four penalty points are given.

If the letters must be underlined but the font doesn't have this attribute, one penalty point is given. The GDI selects the physical font that has the least penalty points. To change the weight of the attributes, use the SetFontMapperWeight function.

## Example of font tools



The following application creates a new font and outputs information about the current physical font.

### New messages

WM\_CREATE

### New functions

CreateFont

GetDC

GetTextMetrics

ReleaseDC

### New structures

TEXTMETRIC

### Brief description

Sent when a window is being created

### Brief description

Creates a logical font

Obtains the device context handle

Fills buffer with values of the current font

Releases the device context handle

### Brief description

Structure of a physical font

## Source code: FONT.C

```
/** FONT.C ****
/* Displays font specifics in default and Times Roman fonts      */
/***** */

#include "windows.h"                                // Include windows.h header file
#include "string.h"                                 // Include string.h header file

long FAR PASCAL FontWndProc ( HWND, unsigned, WORD, LONG);
BOOL FontInit ( HANDLE );

int PASCAL WinMain ( hInstance, hPrevInstance, lpszCmdLine, cmdShow )
HANDLE hInstance, hPrevInstance;                  // Current & previous instance
LPSTR lpszCmdLine;                            // Long ptr to string after
                                              // program name during execution
int cmdShow;                                  // Specifies the application
                                              // window's appearance
{
    MSG msg;                                    // Message variable
    HWND hWnd;                                 // Window handle

    if (!hPrevInstance)                         // Initialize first instance
    {
        if (!FontInit( hInstance )) // If initialization fails
            return FALSE;                      // return FALSE
    }
}

/** Specify appearance of application's main window ****

hWnd = CreateWindow("Font",           // Window class name
                    "Font",          // Window caption
                    WS_OVERLAPPEDWINDOW, // Overlapped window style
                    CW_USEDEFAULT,    // Default upper-left x pos.
                    0,                // Upper-left y pos.
                    CW_USEDEFAULT,    // Default initial x size
                    0,                // y size
                    NULL,             // No parent window
                    NULL,             // Window menu used
                    hInstance,         // Application instance
                    NULL);            // No creation parameters

ShowWindow( hWnd, cmdShow );           // Make window visible
UpdateWindow( hWnd );                 // Update window

while (GetMessage(&msg, NULL, 0, 0)) // Message reading
{
```

## Output

---

```
        TranslateMessage(&msg);           // Message translation
        DispatchMessage(&msg);          // Send message to Windows
    }

    return msg.wParam;                // Return wParam of last message
}

BOOL FontInit( HANDLE hInstance )           // Initialize instance handle
{

/** Specify window class *****/
    WNDCLASS    wcFontClass;           // Main window class

    wcFontClass.style      = CS_VREDRAW | CS_HREDRAW; // Horizontal and
                           // vertical redraw of client area
    wcFontClass.hCursor     = LoadCursor( NULL, IDC_ARROW );;
                           // Mouse cursor
    wcFontClass.hIcon       = LoadIcon( NULL, IDI_APPLICATION );
                           // Default icon
    wcFontClass.lpszMenuName = NULL;           // No menu
    wcFontClass.lpszClassName = "Font";         // Window class
    wcFontClass.hbrBackground = GetStockObject( WHITE_BRUSH );
                           // White background
    wcFontClass.hInstance    = hInstance;        // Instance
    wcFontClass.lpfnWndProc   = FontWndProc;       // Window function
    wcFontClass.cbClsExtra    = 0;              // No extra bytes
    wcFontClass.cbWndExtra    = 0;              // No extra bytes

    if (!RegisterClass( (LPWNDCLASS) &wcFontClass ) )
                           // Register window class
        return FALSE;           // Return FALSE if registration fails
    return TRUE;            // If registration is successful, return TRUE
}

/** FontWndProc *****/
/** Main window function: All messages are sent to this window ***/
*****
```

```
long FAR PASCAL FontWndProc( hWnd, message, wParam, lParam )
HWND    hWnd;                      // Window handle
unsigned message;                 // Message type
WORD    wParam;                   // Message-dependent 16 bit value
LONG    lParam;                   // Message-dependent 32 bit value
{
    static TEXTMETRIC tm, tmNew;
    static short      xChar, yChar, xCharNew, yCharNew;
```

```
PAINTSTRUCT          ps;                      // Paint structure
HDC                 hDC;                     // Device context handle
int                i;
static char         chtmName[20][30];
HFONT               hFontNew, hFontOld; // Fonts

switch (message)           // Process messages
{
    case WM_CREATE:        // Create window
        hDC = GetDC (hWnd);
        GetTextMetrics (hDC, &tm);
        xChar = tm.tmAveCharWidth;
        yChar = tm.tmHeight + tm.tmExternalLeading;
        ReleaseDC (hWnd, hDC);
        break;

    case WM_PAINT:          // Client area redraw
        hdc = BeginPaint( hWnd, &ps );
}

/** Print font parameters in default font *****/
wsprintf( chtmName[0], "tmHeight =             %d",
          tm.tmHeight);
wsprintf( chtmName[1], "tmAscent =            %d",
          tm.tmAscent);
wsprintf( chtmName[2], "tmDescent =           %d",
          tm.tmDescent);
wsprintf( chtmName[3], "tmInternalLeading = %d",
          tm.tmInternalLeading);
wsprintf( chtmName[4], "tmExternalLeading = %d",
          tm.tmExternalLeading);
wsprintf( chtmName[5], "tmAveCharWidth =      %d",
          tm.tmAveCharWidth);
wsprintf( chtmName[6], "tmMaxCharWidth =      %d",
          tm.tmMaxCharWidth);
wsprintf( chtmName[7], "tmWeight =             %d",
          tm.tmWeight);
wsprintf( chtmName[8], "tmItalic =             %d",
          tm.tmItalic);
wsprintf( chtmName[9], "tmUnderlined =         %d",
          tm.tmUnderlined);
wsprintf( chtmName[10], "tmStruckOut =          %d",
          tm.tmStruckOut);
wsprintf( chtmName[11], "tmFirstChar =           %xH",
          tm.tmFirstChar);
wsprintf( chtmName[12], "tmLastChar =            %xH",
          tm.tmLastChar);
```

## Output

---

```
wsprintf( chtmName[13], "tmDefaultChar =      %xH",
          tm.tmDefaultChar);
wsprintf( chtmName[14], "tmBreakChar =      %xH",
          tm.tmBreakChar);
wsprintf( chtmName[15], "tmPitchAndFamily =   %xH",
          tm.tmPitchAndFamily);
wsprintf( chtmName[16], "tmCharSet =      %xH",
          tm.tmCharSet);
wsprintf( chtmName[17], "tmOverhang =      %d",
          tm.tmOverhang);
wsprintf( chtmName[18], "tmDigitizedAspectX = %d",
          tm.tmDigitizedAspectX);
wsprintf( chtmName[19], "tmDigitizedAspectY = %d",
          tm.tmDigitizedAspectY);

for (i = 0; i < 20; i++)
    TextOut( hDC, xChar, yChar * (i+1), chtmName[i],
              strlen(chtmName[i]));

/* Generate new font *****/
hFontNew = CreateFont( 12,                  // lfHeight
                      8,                   // lfWidth
                      0,                   // lfEscapement
                      0,                   // lfOrientation
                      700,                 // lfWeight
                      FALSE,                // lfItalic
                      TRUE,                 // lfUnderline
                      FALSE,                // lfStrikeOut
                      OEM_CHARSET,          // lfCharset
                      OUT_DEFAULT_PRECIS,
                      // lfOutPrecision
                      CLIP_DEFAULT_PRECIS,
                      // lfClipPrecision
                      DEFAULT_QUALITY, // lfQuality
                      VARIABLE_PITCH | FF_ROMAN,
                      // lfPitchAndFamily
                      "TmsRmn");           // lfFaceName
hFontOld = SelectObject( hDC, hFontNew);

GetTextMetrics (hDC, &tmNew);
xCharNew = tmNew.tmAveCharWidth;
yCharNew = tmNew.tmHeight + tmNew.tmExternalLeading;

/* Print font parameters in new font *****/
wsprintf( chtmName[0], "tmHeight =      %d",
          tmNew.tmHeight);
```

```
wsprintf( chtmName[1], "tmAscent = %d",
          tmNew.tmAscent);
wsprintf( chtmName[2], "tmDescent = %d",
          tmNew.tmDescent);
wsprintf( chtmName[3], "tmInternalLeading = %d",
          tmNew.tmInternalLeading);
wsprintf( chtmName[4], "tmExternalLeading = %d",
          tmNew.tmExternalLeading);
wsprintf( chtmName[5], "tmAveCharWidth = %d",
          tmNew.tmAveCharWidth);
wsprintf( chtmName[6], "tmMaxCharWidth = %d",
          tmNew.tmMaxCharWidth);
wsprintf( chtmName[7], "tmWeight = %d",
          tmNew.tmWeight);
wsprintf( chtmName[8], "tmItalic = %xH",
          tmNew.tmItalic);
wsprintf( chtmName[9], "tmUnderlined = %xH",
          tmNew.tmUnderlined);
wsprintf( chtmName[10], "tmStruckOut = %xH",
          tmNew.tmStruckOut);
wsprintf( chtmName[11], "tmFirstChar = %xH",
          tmNew.tmFirstChar);
wsprintf( chtmName[12], "tmLastChar = %xH",
          tmNew.tmLastChar);
wsprintf( chtmName[13], "tmDefaultChar = %xH",
          tmNew.tmDefaultChar);
wsprintf( chtmName[14], "tmBreakChar = %xH",
          tmNew.tmBreakChar);
wsprintf( chtmName[15], "tmPitchAndFamily = %xH",
          tmNew.tmPitchAndFamily);
wsprintf( chtmName[16], "tmCharSet = %xH",
          tmNew.tmCharSet);
wsprintf( chtmName[17], "tmOverhang = %d",
          tmNew.tmOverhang);
wsprintf( chtmName[18], "tmDigitizedAspectX = %d",
          tmNew.tmDigitizedAspectX);
wsprintf( chtmName[19], "tmDigitizedAspectY = %d",
          tmNew.tmDigitizedAspectY);

for (i = 0; i < 20; i++)
    TextOut( hDC, xCharNew + 250, yCharNew * (i+1),
              chtmName[i], strlen(chtmName[i]));

DeleteObject( SelectObject( hDC, hFontOld));
EndPaint( hWnd, &ps );
break;
```

## *Output*

---

```
        case WM_DESTROY:           // Destroy window
            PostQuitMessage(0);
            break;

        default:
            return (DefWindowProc( hWnd, message, wParam, lParam ));
            break;
    }
return(0L);
}
```

## Module definition file: FONT.DEF

```
NAME      Font
DESCRIPTION 'Font example'
EXETYPE   WINDOWS
STUB      'WINSTUB.EXE'
CODE      PRELOAD MOVEABLE
DATA      PRELOAD MOVEABLE MULTIPLE
HEAPSIZE  4096
STACKSIZE 4096
EXPORTS   FontWndProc @1
```

To compile and link this application, use the COMPILE.BAT batch file described earlier in this book. You can create this file using the DOS COPY CON command or any text editor or word processor that generates ASCII files. Here's the listing again:

```
cl -c -Gw -Zp %1
link /align:16 %1,%1.exe,,libw+slibcew,%1.def
rc %1.exe
```

Save this file to a directory containing your path or one containing the source and definition files. Type the following and press **Enter**:

```
compile font
```

The batch file performs all the tasks needed.

## How FONT.EXE works

When the window is being created, the WM\_CREATE message occurs. This is the first message to reach the window procedure. A device context, which contains the system font as a default, is obtained. The GetTextMetrics function places information about this font in the tm variable of the TEXTMETRIC structure.

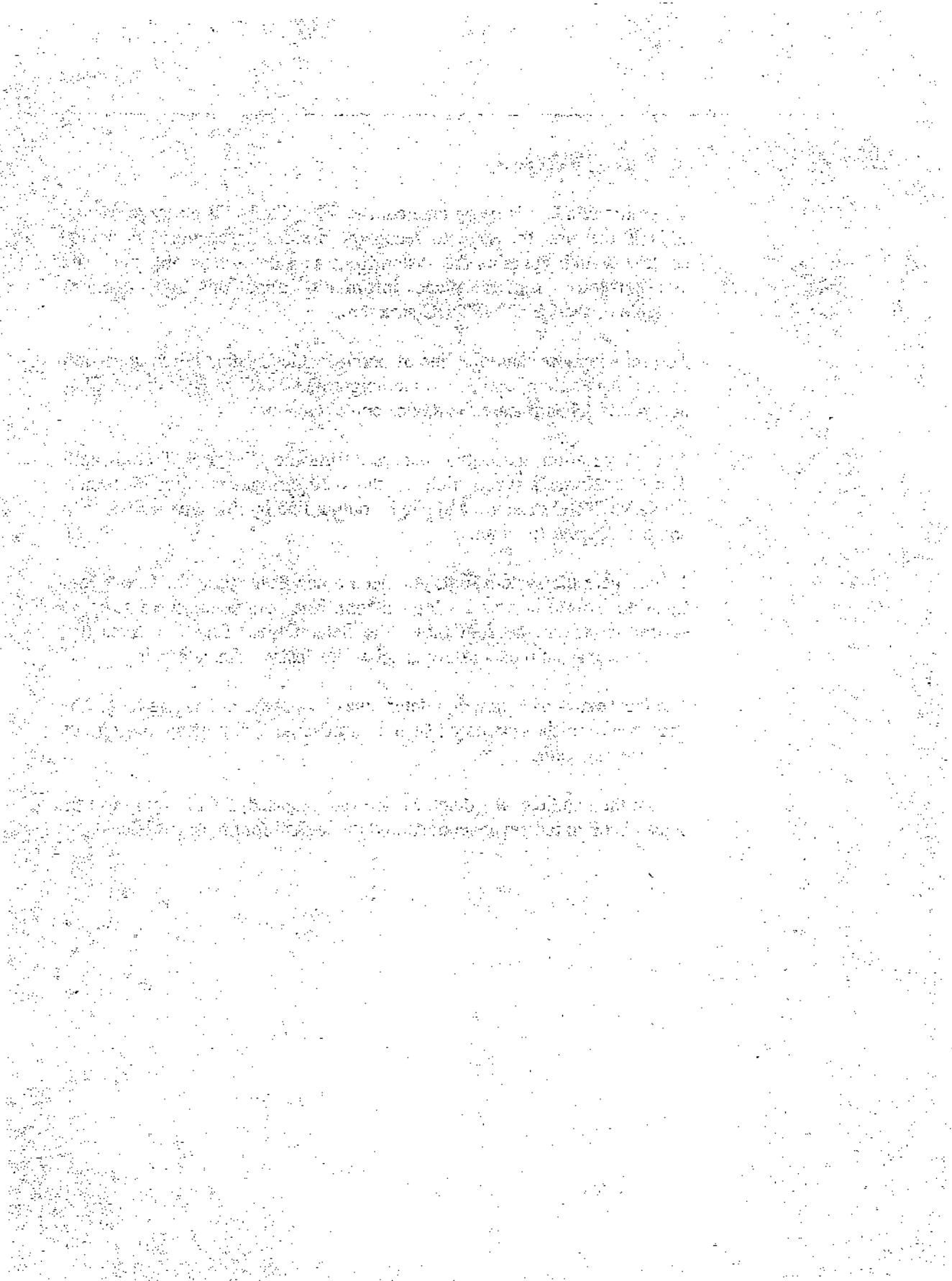
To produce more than one line of output, which begins in the upper-left corner of the client area, save the height and width of a letter in xChar and yChar. Then release the device context again.

Text preparation and output occurs during the WM\_PAINT message. The chtmName array, which contains 20 strings, is filled from the TEXTMETRIC structure and can be output line by line with a FOR loop and the TextOut function.

To compare the system font, you need a new font. First the CreateFont function is used to create a logical font. The parameters were selected randomly. From the font table, the SelectObject function takes the physical font that most closely matches the desired characteristics.

The GetTextMetrics function determines how these characters look. The characters are then prepared in the chtmName string array and output with the new font.

Before the handle of the display context is released, the old font must be placed back in the device context and the logical font must be deleted.



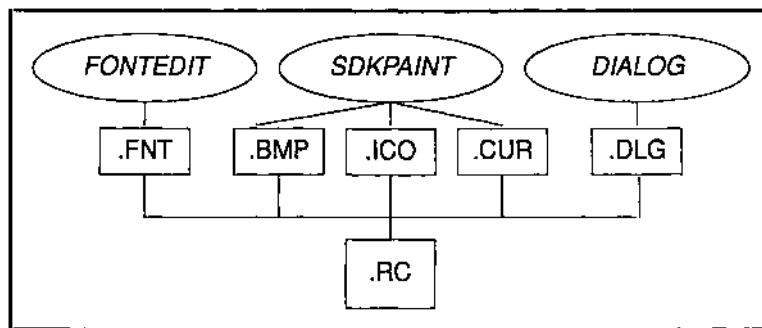
# Resources

## Overview

In addition to code and data segments, a Windows application can also have resource segments, which are also located in the .EXE file.

Resources are data that usually cannot be changed while an application is running. Similar to code segments, resource segments are only present once. So, several instances of the same application share the resources. There are nine types of resources:

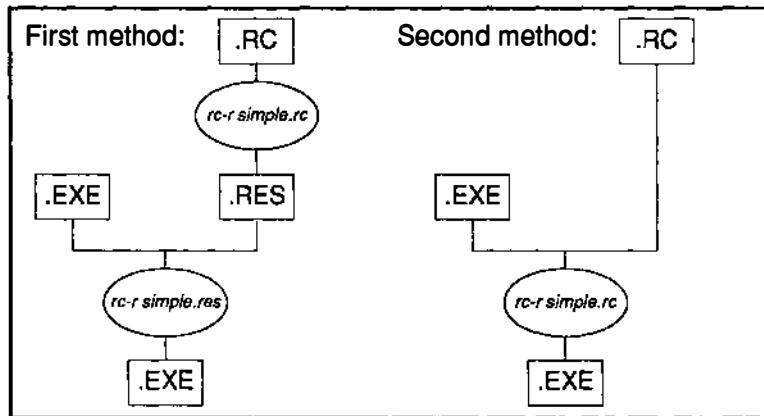
- Keyboard accelerator
- User-defined resource
- Bitmap
- Cursor
- Dialog box
- Font
- Icon
- Menu
- String table



These resources are described in a separate file called the resource script. This is an ASCII text file with the .RC extension that can refer to different files (see above illustration).

A special compiler, called the resource compiler (RC.EXE), translates the .RC file into binary format, inserts it in the .EXE file, and enters references in the resource table (the header area of the application).

There are two ways to use the resource compiler:



The first method involves two separate programming steps. This is the preferred method because the compiler uses the -r switch to check for syntax errors and, if necessary, stops the program. This method then adds the binary format (which is placed in a .RES file) to the .EXE file.

It's also possible to place menus, dialog boxes, etc. in the data segment. However, if you have your own resource script, it's easier to exchange the resources that are defined in this file. For example, to exchange an English menu for a Spanish menu in an application, simply replace the English resource script with the Spanish resource script and start the resource compiler.

You don't have to run a C compiler or linker. The linker is needed only if the menus in both .RC files have different names. This is necessary because the resource compiler cannot remove old resources from the resource table. The resource compiler uses a preprocessor called RCPP.EXE, which, like the C preprocessor, is familiar with the #include and #define statements.

The resources can be divided into two groups:

- Single-line statements (e.g., CURSOR)
- Multiple-line statements (e.g., MENU)

In the first group (single-line statements), the .RC file contains a reference to another file, instead of containing the actual resource data.

This other file could have been created by another application, such as the SDKPAINT utility. These files are usually in binary format.

The first time the resource compiler is called, these files are placed in the .RES file by the -r switch. However, the resources of the second group (multiple-line statements) are defined entirely in the resource script file.

The following is the structure of resources in the .RC file (with one exception):

Name	Type	Data
------	------	------

You should clearly understand this structure after we discuss the individual resources in the following sections.

For all single-line statements, two options can precede the data specification. These options include a load option (PRELOAD or LOADONCALL) and a save option (FIXED, MOVEABLE, and DISCARDABLE).

The default settings are LOADONCALL, MOVEABLE, AND DISCARDABLE. However, the DISCARDABLE option isn't used with bitmaps.

The resource must be loaded in the source text in order to use it. Each resource (except user-defined resources) has its own function, which loads the resource and retrieves a handle.

## Icons

An icon is a small graphic that represents a minimized application and is also used in message and dialog boxes. Icons can also be drawn in the client area.

There are three ways to display icons in the main window. The desired display option is specified in the WNDCLASS structure so that all windows that refer to the same class have the same icon.

One of five predefined icons are used. You can address these icons by using the following names:

IDI_APPLICATION	Default icon; empty rectangle
IDI_ASTERISK	Icon with small asterisk
IDI_EXCLAMATION	Icon with small exclamation point
IDI_HAND	Icon with stop sign
IDI_QUESTION	Icon with question mark

In the WNDCLASS structure, you're prompted for a handle to the icon. To obtain this handle, use the LoadIcon function. This function has two parameters. The first parameter must be set to NULL for predefined icons. In the second parameter, specify one of the five ID values listed above.

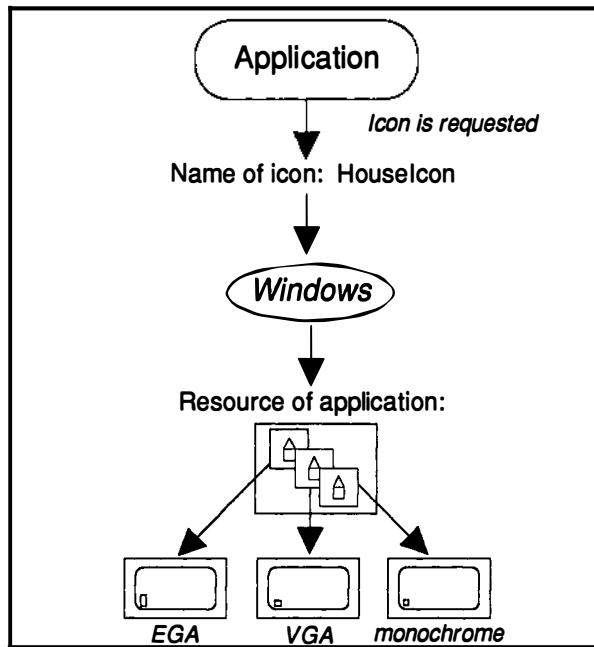
We used the predefined icon, IDI\_APPLICATION, for all the applications created in the previous chapters.

Usually a predefined icon isn't assigned to an application. Instead, you must create an icon that clearly represents the application so that the user can immediately identify the application by its icon.

You can create icons by using the SDKPAINT.EXE utility, which is part of the Software Development Kit.

After creating an icon, save it in its own .ICO file, which contains several graphics in bitmap form. Although these graphics look similar, they have been adapted for different screen resolutions.

When the application calls the icon, it uses the name defined in the .RC file. Windows then decides which of the icons best fits the current screen. So, the application doesn't have to consider the screen type.



The .ICO file is defined in the .RC file with a single-line statement:

Name	Type	Data
HouseIcon	ICON	house.ico

Specify the type with the keyword ICON. You can use any name you want for the icon because it is set as the second parameter in the LoadIcon function. The name is passed as a string. For all the icons that you create, the first parameter is the handle to the current instance (hInstance).

```
wcIconClass.hIcon = LoadIcon( hInstance, "HouseIcon" );
```

Instead of identifying an icon with a name, you can also use a number ID.

Resource Script File:	234 ICON house.ico
Source File:	hIcon = LoadIcon( hInstance, MAKEINTRESOURCE(234));

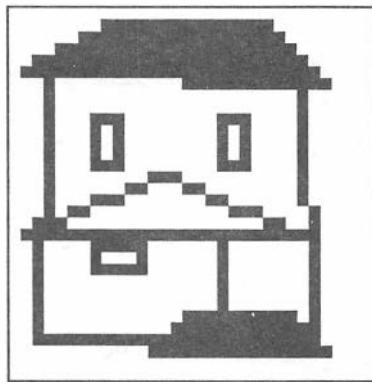
The MAKEINTRESOURCE macro converts an integer value into a far pointer. This macro also defines ID values or names of predefined icons,

such as IDI\_APPLICATION in WINDOWS.H. You should use this method when numerous icons are set in the .RC file. This makes the .EXE file smaller and the LoadIcon function execute more quickly.

During the runtime of the application, you can replace the icon of a window class with another icon that is also defined in the .RC file and whose handle was also obtained with LoadIcon. The SetClassWord function can change certain fields of the WNDCLASS structure after the class is registered, including the handle of the icon.

If the specified icon is already in memory, the LoadIcon function only obtains a handle to this object, but doesn't reload it. This also applies to several instances of a Windows application.

## Example of icon access from an RC file



Here's a basic example of icon access using a resource script and the above icon. Starting with this listing, this and subsequent source code listings in this book will be commented in less detail than those appearing in preceding chapters. We'll add commentary to the most important lines. Refer to earlier listings for details on most uncommented routines.

### New functions

LoadIcon

### Brief description

Loads an icon and retrieves the handle

## Source code: ICON1.C

```
/** ICON1.C ****
** Appears in window (or minimized) as icon, with the icon specified by **
** the HOUSE.ICO icon.
****

#include "windows.h"                                // Include windows.h header file

long FAR PASCAL Icon1WndProc ( HWND, unsigned, WORD, LONG);
BOOL Icon1Init ( HANDLE );

int PASCAL WinMain ( hInstance, hPrevInstance, lpszCmdLine, cmdShow )
HANDLE  hInstance, hPrevInstance;                  // Current & previous instance
LPSTR   lpszCmdLine;                            // Long ptr to string after
                                                // program name during execution
int      cmdShow;                             // Specifies the application
                                                // window's appearance
{
    MSG      msg;                           // Message variable
    HWND     hWnd;                          // Window handle

    if (!hPrevInstance)                    // Initialize first instance
    {
        if (!Icon1Init( hInstance ))
            return FALSE;
    }
}

/** Specify appearance of application's main window ****

hWnd = CreateWindow("Icon1",
                    "Icon1",
                    WS_OVERLAPPEDWINDOW,
                    CW_USEDEFAULT, 0,
                    CW_USEDEFAULT, 0,
                    NULL,
                    NULL,
                    hInstance,
                    NULL);

ShowWindow( hWnd, cmdShow );                      // Make window visible
UpdateWindow( hWnd );                           // Update window

while (GetMessage(&msg, NULL, 0, 0)) // Message reading
{
    TranslateMessage(&msg);                // Message translation
    DispatchMessage(&msg);                // Send message to Windows
}
```

## Resources

---

```
}

    return msg.wParam;                                // Return wParam of last message
}

BOOL Icon1Init( HANDLE hInstance )
{
    . .

/** Specify window class *****/
    WNDCLASS      wcIcon1Class;

    wcIcon1Class.style      = CS_VREDRAW | CS_HREDRAW;
    wcIcon1Class.hCursor     = LoadCursor( NULL, IDC_ARROW );
    wcIcon1Class.hIcon       = LoadIcon( hInstance, "HouseIcon");

                                            // Load HouseIcon resource

    wcIcon1Class.lpszMenuName = NULL;
    wcIcon1Class.lpszClassName = "Icon1";
    wcIcon1Class.hbrBackground = GetStockObject( WHITE_BRUSH );
    wcIcon1Class.hInstance    = hInstance;
    wcIcon1Class.lpfnWndProc  = Icon1WndProc;
    wcIcon1Class.cbClsExtra   = 0;
    wcIcon1Class.cbWndExtra   = 0;

    if (!RegisterClass( (LPWNDCLASS) &wcIcon1Class ) )
        return FALSE;
    return TRUE;
}

/** Icon1WndProc *****/
/** Main window function: All messages are sent to this window */
/** *****/
long FAR PASCAL Icon1WndProc( hWnd, message, wParam, lParam )
HWND hWnd;
unsigned message;
WORD wParam;
LONG lParam;
{
    switch (message)
    {
        case WM_DESTROY:                      // Destroy window
            PostQuitMessage(0);
            break;
    }
}
```

---

```

        default:
            return (DefWindowProc( hWnd, message, wParam, lParam ));
        break;
    }
return(0L);
}

```

## Module definition file: ICON1.DEF

```

NAME      Icon1

DESCRIPTION 'Icon from resource'

EXETYPE   WINDOWS

STUB      'WINSTUB.EXE'

CODE      PRELOAD MOVEABLE
DATA      PRELOAD MOVEABLE MULTIPLE

HEAPSIZE  4096
STACKSIZE 4096

EXPORTS   Icon1WndProc @1

```

## Resource script: ICON1.RC

```
HouseIcon      ICON      house.ico
```

As mentioned earlier in this chapter, applications using separate resources require extra steps in compiling and linking. The steps are as follows:

- 1) Compile the source code as in earlier examples. Thus, compiling the ICON1.C source code would require the following line:

```
cl -c -Gw -Zp icon1.c
```

- 2) Call the resource compiler with the -r switch to prepare the .RC file. The following line compiles the ICON1.RC file:

```
rc -r icon1.rc
```

- 3) Link the source code to the module definition file, generating an executable file without resources. The following line links the ICON1.OBJ and ICON1.DEF files to make ICON1.EXE without resources:

```
link /align:16 icon1,icon1.exe,,libw+slibcew,icon1.def
```

- 4) Call the resource compiler again to add resource references. The following line completes the task:

```
rc icon1.res
```

You can compile and link this application faster by creating a batch file named RCOMPILE.BAT. Create this file using the DOS COPY CON command or any text editor or word processor that generates ASCII files. Type the following to create RCOMPILE.BAT:

```
cl -c -Gw -Zp %1  
rc -r %1.rc  
link /align:16 %1,%1.exe,,libw+slibcew,%1.def  
rc %1.res
```

Save this file to a directory contained in your path or one containing the source, module definition and resource scripts. Type the following and press **[Enter]** to compile ICON1:

```
rcompile icon1
```

The batch file performs all the tasks needed.

## How ICON1.EXE works

This program's only task is to display a custom icon when the window is minimized.

This icon was drawn by SDKPAINT.EXE and represents a house. The name "HouseIcon" was assigned to the icon in the .RC file. This name was also specified in the LoadIcon function.

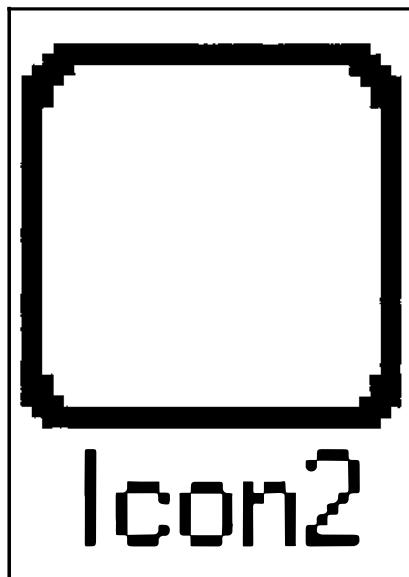
Icons that were defined in an .RC file cannot be used for dynamic displays. For example, in the CLOCK application the hands move in the minimized icon. In this case, the hIcon field is set to NULL in the

WNDCLASS structure. This informs Windows that it should also send WM\_PAINT messages to the window procedure when the window is minimized.

When this message is being processed, the IsIconic function asks whether the window is currently being displayed as an icon. If the result is true, you can draw icons from within the application with graphic functions.

## Example of dynamic icon access

The following application generates a minimized icon from graphic functions. Here's how the icon appears when you minimize the application:



### New functions

IsIconic

### Brief description

TRUE, if the window is being displayed as an icon

## Source code: ICON2.C

```
/** ICON2.C ****
/** Generates minimized icon dynamically from within program ****
****/
```

## Resources

---

```
#include "windows.h"                                // Include windows.h header file

long FAR PASCAL Icon2WndProc ( HWND, unsigned, WORD, LONG);
BOOL Icon2Init ( HANDLE );

int PASCAL WinMain ( hInstance, hPrevInstance, lpszCmdLine, cmdShow )
HANDLE  hInstance, hPrevInstance;                  // Current & previous instance
LPSTR   lpszCmdLine;                            // Long ptr to string after
                                                // program name during execution
int     cmdShow;                               // Specifies the application
                                                // window's appearance
{
    MSG msg;                                 // Message variable
    HWND hWnd;                               // Window handle

    if (!hPrevInstance)                      // Initialize first instance
    {
        if (!Icon2Init( hInstance ))
            return FALSE;
    }

    /* Specify appearance of application's main window *****/
    hWnd = CreateWindow("Icon2",
                        "Icon2",
                        WS_OVERLAPPEDWINDOW,
                        CW_USEDEFAULT, 0,
                        CW_USEDEFAULT, 0,
                        NULL,
                        NULL,
                        hInstance,
                        NULL);

    ShowWindow( hWnd, cmdShow );           // Make window visible
    UpdateWindow( hWnd );                // Update window

    while (GetMessage(&msg, NULL, 0, 0)) // Message reading
    {
        TranslateMessage(&msg);          // Message translation
        DispatchMessage(&msg);          // Send message to Windows
    }

    return msg.wParam;                   // Return wParam of last message
}

BOOL Icon2Init( HANDLE hInstance )
```

```
{  
  
    /** Specify window class *****/  
  
    WNDCLASS    wcIcon2Class;  
  
    wcIcon2Class.style      = CS_VREDRAW | CS_HREDRAW;  
    wcIcon2Class.hCursor    = LoadCursor( NULL, IDC_ARROW );  
    wcIcon2Class.hIcon      = NULL; // Note - no icon  
    wcIcon2Class.lpszMenuName = NULL;  
    wcIcon2Class.lpszClassName = "Icon2";  
    wcIcon2Class.hbrBackground = GetStockObject( WHITE_BRUSH );  
    wcIcon2Class.hInstance   = hInstance;  
    wcIcon2Class.lpfnWndProc = Icon2WndProc;  
    wcIcon2Class.cbClsExtra  = 0;  
    wcIcon2Class.cbWndExtra  = 0;  
  
    if (!RegisterClass( (LPWNDCLASS) &wcIcon2Class ) )  
        return FALSE;  
    return TRUE;  
}  
  
/** Icon2WndProc *****/  
/** Main window function: All messages are sent to this window */  
/** *****/  
  
long FAR PASCAL Icon2WndProc( hWnd, message, wParam, lParam )  
HWND    hWnd;  
unsigned message;  
WORD    wParam;  
LONG    lParam;  
{  
    PAINTSTRUCT    ps;  
    HDC             hdc;  
    HPEN            hPen, hPenOld;  
  
    switch (message)  
    {  
        case WM_PAINT:           // Client area redraw  
            hdc = BeginPaint( hWnd, &ps );  
            if (IsIconic(hWnd)) // Minimized? Draw icon  
            {  
                hPen = CreatePen(PS_SOLID, 3, RGB(0,0,0));  
                // Solid pen, black color  
                hPenOld = SelectObject(hdc, hPen);  
                RoundRect( hdc, ps.rcPaint.left, ps.rcPaint.top,  
                           ps.rcPaint.right,
```

```
    ps.rcPaint.bottom, (ps.rcPaint.right -  
ps.rcPaint.left)/3,  
                           (ps.rcPaint.bottom - ps.rcPaint.top)/2);  
                           // Draw rounded rectangle when  
                           // minimized icon is needed  
                           DeleteObject(SelectObject(hDC, hPenOld));  
}  
EndPaint( hWnd, &ps);  
break;  
  
case WM_DESTROY:           // Destroy window  
    PostQuitMessage(0);  
    break;  
  
default:  
    return (DefWindowProc( hWnd, message, wParam, lParam ));  
break;  
}  
return(0L);  
}
```

## Module definition file: ICON2.DEF

```
NAME      Icon2  
  
DESCRIPTION 'Dynamic icons'  
  
EXETYPE   WINDOWS  
  
STUB      'WINSTUB.EXE'  
  
CODE      PRELOAD MOVEABLE  
DATA      PRELOAD MOVEABLE MULTIPLE  
  
HEAPSIZE  4096  
STACKSIZE 4096  
  
EXPORTS   Icon2WndProc  @1
```

To compile and link this application, use the COMPILE.BAT batch file described earlier in this book. Here's the listing again:

```
cl -c -Gw -Zp %1  
link /align:16 %1,%1.exe,,libw+slibcew,%1.def  
rc %1.exe
```

Save this file to a directory contained in your path or one containing your source and definition files. Type the following and press **Enter**:

```
compile icon2
```

The batch file performs all the tasks needed.

## How ICON2.EXE works

The icon for this application isn't created until the application is executed. Since the value for hIcon is set to NULL in the WNDCLASS structure, the WM\_PAINT messages also appear when the window is minimized.

The IsIconic function prompts for the status of the window. If the return value is TRUE, then ICON2 draws a rounded rectangle with a font width of 3.

## Cursor

The cursor usually displays mouse movements on the screen. Similar to icons, various options can be used to display different mouse pointers.

Besides the default arrow cursor, you can also use other predefined .cursor types:

IDC_ARROW	Default arrow cursor
IDC_CROSS	Cursor as a cross
IDC_IIBEAM	Cursor as an I-beam for a text
IDC_ICON	Cursor as a rectangle with an inner rectangle
IDC_SIZE	Cursor as a cross with four arrows
IDC_SIZENESW	Cursor as a double arrow in direction: NE->SW
IDC_SIZENS	Cursor as a double arrow in direction: N->S
IDC_SIZENWSE	Cursor as a double arrow in direction: NW->SE
IDC_SIZEWE	Cursor as a double arrow in direction: W->E
IDC_UPARROW	Cursor as a vertical arrow
IDC_WAIT	Cursor as an hourglass

The LoadCursor function obtains the handle of a predefined cursor. The first parameter is set to NULL and the second parameter contains one of the ID numbers we previously mentioned.

A field exists for this cursor handle in the WNDCLASS structure just as there is a field for the icon handle. If you specify the cursor here, then it applies for all windows that access this class, even if they are in different instances.

Whenever the application begins executing a long operation (e.g., loading data from the hard drive), the user should be informed of this using the hourglass icon.

Also, you should retain the mouse capture so that the operation isn't destroyed by mouse actions. Once the operation is complete, the old cursor is restored and the capture is released.

```
HCURSOR hHourglass, hAltCursor
hHourglass = LoadCursor(NULL, IDC_WAIT);
→SetCapture(hWnd);
→hAltCursor = SetCursor(hHourglass)
longer lasting operation
→SetCursor(hAltCursor);
→ReleaseCapture();
```

Use SDKPAINT.EXE to draw your own cursor. While doing this, you can also set a hot spot that specifies which point is the decisive point within the cursor. The files that are created from this operation receive the .CUR extension.

When you use LoadCursor, the instance handle is passed to the first parameter and the name of the cursor is passed to the second parameter as a string, which is also specified in the .RC file.

```
/** As it appears in the resource script ***/
MouseCursor      CURSOR      mouse.cur

/** As it appears in the source code *****/
wcCursorClass.hCursor = LoadCursor( hInstance, "MouseCursor" );
```

Since the NULL value is set in the WNDCLASS structure instead of the LoadCursor call, the cursor doesn't change its shape when it enters the client area. Unfortunately, this has a negative effect for the user (e.g., a double arrow may be displayed in the client area). So you should use this method only when the cursor is set in the window function.

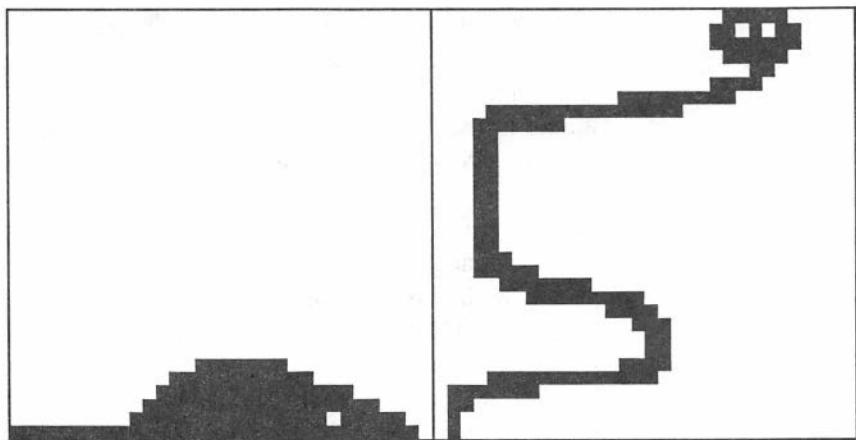
For example, use this method during runtime if you want the cursor shape to change on the basis of certain events. First you must reload the cursor with LoadCursor and then make it the current cursor with SetCursor. This setting should occur during the WM\_MOUSEMOVE message.

You shouldn't specify the cursor in the WNDCLASS structure and then set a different cursor in the WM\_MOUSEMOVE message. This will cause the mouse pointer to flicker.

Setting a cursor usually applies only to the client area. If the cursor is located above the Control menu or the border, Windows determines the appearance of the cursor unless you use SetCapture to retain the mouse capture. If you do this, the mouse pointer doesn't change when you leave the client area.

## Example using a self-defined cursor

This example doesn't contain a class cursor. Instead, different mouse pointers are set when the mouse is moved. Which cursor is current depends on the mouse button being pressed. Three mouse pointers are used (two self-defined and one predefined cursor). Here are the self-defined cursors, named MOUSE.CUR and SNAKE.CUR:



**New functions**

GetWindowWord  
LoadCursor  
SetCursor

**Brief description**

Determines current instance  
Obtains cursor handle  
Changes cursor shape

## Source code: CURSOR.C

```
/** CURSOR.C ****
/** Accesses three different types of cursors: Up arrow (a default),      */
/** mouse and snake (generated using SDKPAINT.EXE)                      */
/** ****
#include "windows.h"                                     // Include windows.h header file

BOOL CursorInit ( HANDLE );
long FAR PASCAL CursorWndProc( HWND, unsigned, WORD, LONG);

int PASCAL WinMain( hInstance, hPrevInstance, lpszCmdLine, cmdShow )
HANDLE hInstance, hPrevInstance;                         // Current & previous instance
LPSTR lpszCmdLine;                                     // Long ptr to string after
                                                       // program name during execution
int cmdShow;                                           // Specifies the application
                                                       // window's appearance
{
    MSG msg;                                            // Message variable
    HWND hWnd;                                         // Window handle

    if (!hPrevInstance) {                                // Initialize first instance
        if (!CursorInit( hInstance ))
            return FALSE;
    }
}
```

```
}

/** Specify appearance of application's main window *****/
hWnd = CreateWindow("Cursor",
                     "Cursor",
                     WS_OVERLAPPEDWINDOW,
                     CW_USEDEFAULT, 0, CW_USEDEFAULT, 0,
                     NULL,
                     NULL,
                     hInstance,
                     NULL);

ShowWindow( hWnd, cmdShow );           // Make window visible
UpdateWindow( hWnd );                // Update window

while (GetMessage(&msg, NULL, 0, 0)) // Message reading
{
    TranslateMessage(&msg);          // Message translation
    DispatchMessage(&msg);          // Send message to Windows
}

return (int)msg.wParam;               // Return wParam of last message
}

BOOL CursorInit( hInstance )
HANDLE hInstance;
{
    /** Specify window class *****/
    WNDCLASS      wcCursorClass;

    wcCursorClass.hCursor      =      NULL;
    wcCursorClass.hIcon        =      LoadIcon( NULL, IDI_APPLICATION );
    wcCursorClass.lpszMenuName =      NULL;
    wcCursorClass.lpszClassName =     "Cursor";
    wcCursorClass.hbrBackground =     (HBRUSH)GetStockObject( WHITE_BRUSH );
};

    wcCursorClass.hInstance    =      hInstance;
    wcCursorClass.style       =      CS_VREDRAW | CS_HREDRAW |
CS_DBLCLKS;
    wcCursorClass.lpfnWndProc =      CursorWndProc;
    wcCursorClass.cbClsExtra =      0 ;
    wcCursorClass.cbWndExtra =      0 ;
```

## Resources

---

```
if (!RegisterClass( &wcCursorClass ) )
    return FALSE;

return TRUE;
}

/** CursorWndProc *****/
/** Main window function: All messages are sent to this window      */
/** *****/
long FAR PASCAL CursorWndProc( hWnd, message, wParam, lParam )
HWND      hWnd;
unsigned message;
WORD      wParam;
LONG      lParam;
{
    static BOOL      bCapture = FALSE;
    static HCURSOR   hCur;
    static HANDLE    hInst;

    switch (message)
    {
        case WM_CREATE:
            hInst = GetWindowWord( hWnd, GWW_HINSTANCE);
            break;

        case WM_LBUTTONDOWNCLK:
            if (bCapture)
            {
                ReleaseCapture();
                bCapture = FALSE;
            }
            else
            {
                SetCapture(hWnd);
                bCapture = TRUE;
            }
            break;

        case WM_MOUSEMOVE:           // Change cursor appearance
                                    // during mouse movement,
                                    // depending on which mouse
                                    // button is pressed
            switch (wParam)
            {
                case MK_LBUTTON: // Display mouse cursor
                                // if left button is pressed
                                // during movement

```

```
        hCur = LoadCursor(hInst, "MouseCursor");
        break;
    case MK_RBUTTON: // Display snake cursor
                      // if right button is pressed
                      // during movement
        hCur = LoadCursor(hInst, "SnakeCursor");
        break;
    default:          // Load up arrow as default
        hCur = LoadCursor(NULL, IDC_UPARROW);
        break;
    }
    SetCursor(hCur);
    break;

case WM_DESTROY:           // Destroy window
    PostQuitMessage(0);
    break;

default:
    return (DefWindowProc( hWnd, message, wParam, lParam ));
break;
}
return(0L);
}
```

## Module definition file: CURSOR.DEF

NAME	Cursor
DESCRIPTION	'Cursor demonstration'
EXETYPE	WINDOWS
STUB	'WINSTUB.EXE'
CODE	PRELOAD MOVEABLE
DATA	PRELOAD MOVEABLE MULTIPLE
HEAPSIZE	4096
STACKSIZE	4096
EXPORTS	CursorWndProc @1

## Resource script: CURSOR.RC

```
MouseCursor  CURSOR  mouse.cur
SnakeCursor  CURSOR  snake.cur
```

To compile and link this application, use the RCOMPILE.BAT batch file described earlier in this chapter. Here's the listing again:

```
cl -c -Gw -Zp %1
rc -r %1.rc
link /align:16 %1,%1.exe,,libw+slibcew,%1.def
rc %1.res
```

Save this file to the directory containing your source, module definition and resource scripts. Type the following and press **Enter** to compile CURSOR:

```
rcompile cursor
```

The batch file performs all the tasks needed.

## How CURSOR.EXE works

The wcCursorClass.hCursor field in the WNDCLASS structure is set to NULL. Since the LoadCursor function in the window procedure requires the handle of the current instance, the handle is determined with the GetWindowWord function in the WM\_CREATE message.

You could also determine the handle using lParam, which contains a pointer to a structure containing hInstance. If you move the mouse into the client area while holding down the left mouse button, the cursor displays a mouse.

The mouse turns into a snake when the user presses the right mouse button.

The predefined cursor, containing the IDC\_UPARROW ID value, appears when the mouse button isn't pressed.

When you double-click the left mouse button to retain the mouse capture, the cursor that is currently set applies to the entire screen. If

you double-click again, the mouse capture will be released (refer to the Mouse example section in the chapter on input).

The mouse and snake cursors were created with SDKPAINT.EXE and are entered in the .RC file as cursors.

## Other single-line statements

The two remaining single-line statements control bitmaps and fonts. Instead of being directly defined in the resource script file, they are in separate files linked to statements.

### BITMAP

Bitmaps are pictures 32 pixels wide by 32 pixels high. They are used to draw pictures on the screen (e.g., the system box is a bitmap) and create pattern brushes.

If you use the SDKPAINT application to create bitmaps, the files automatically receive the .BMP extension. In the .RC file the resource type is called BITMAP. Just like icons and the cursor, bitmaps also have a separate loading function called LoadBitmap.

The predefined bitmaps in Windows can be divided into two groups. All the bitmaps whose ID values begin with OBM\_OLD were used in earlier versions of Windows. The other bitmaps have been used since Windows Version 3.0. There are a total of over 30 predefined bitmaps. Bitmaps aren't frequently created until runtime.

### FONT

Use the FONTEdit.EXE utility to create your own fonts. With this editor, you can create the bitmaps on the screen and define the characteristics of the font. The result is a file with the .FNT extension. Use the following procedure to make a font file (e.g., ROMAN.FON) out of this file:

Enter the .FNT file in an .RC file. Here's an example:

There can be several statements here. You must create a blank code segment for the linker. This is usually done in assembly language. Also, the linker needs the definition file containing the following statements: LIBRARY (for defining any name), DESCRIPTION (for describing device specific information), and DATA with the NONE option.

Then the linker can start running. It produces a type of DLL file that has the .EXE extension. You must change the name of this extension to .FON.

## **STRINGTABLE**

This is the first multiple-line statement discussed in this chapter. Use a STRINGTABLE to save character strings that become independent from the data segment. When this happens, these strings can be used with different foreign language versions.

The STRINGTABLE structure in the .RC file is different than the structures of the other resources. Since there can be only one table in an .RC file, the STRINGTABLE doesn't have its own name or ID value.

However, the individual strings, which must be single-line and consist of no more than 255 characters, are identified with an ID value. Unlike the cursor, which has a string for a value, the ID value is a number.

The control characters Tab, Linefeed, and Carriage Return can be specified as octal constants (\011, \012, \015) in the string.

```
#define ID_ERROR_1      13
#define ID_ERROR_2      14
#define ID_ERROR_3      15

STRINGTABLE
BEGIN
    ID_ERROR_1, "File not found"
    ID_ERROR_2, "Write error"
    ID_ERROR_3, "Not enough memory"

END
```

```
LoadString( hInstance, WID, lpBuffer, nBufferMax);  
          ↓    ↓    ↓  
e.g., ID_ERROR_2 e.g., szString e.g., 20
```

The #define statements are optional. You must load one of the strings into a buffer to display or process them by using the LoadString function.

The second parameter specifies the ID value of the desired string. The third parameter points to the buffer where the string is written. The value in the last parameter specifies the number of characters that should be copied to the buffer.

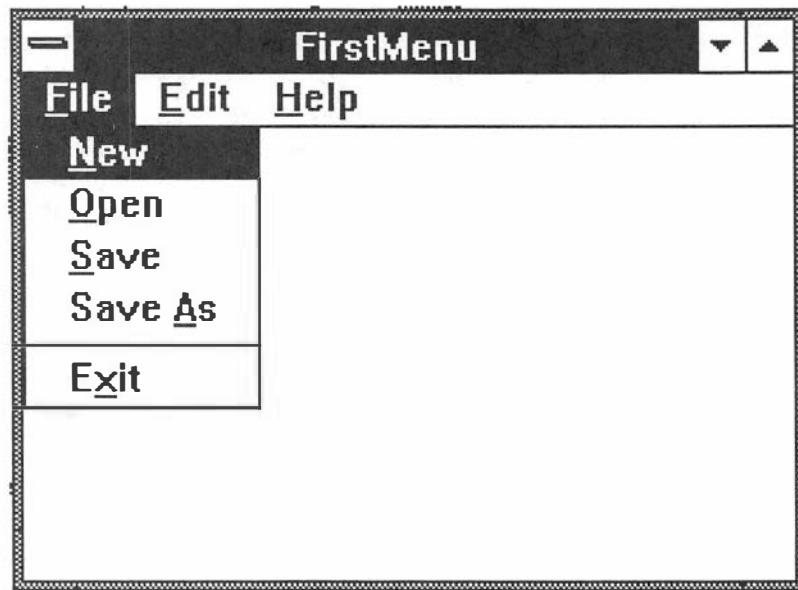
A string that is too long will be truncated. The return value of the function indicates the current number of copied characters in the buffer.

## Menus

A menu consists of various items that are displayed in one or more lines beneath the title bar of a window. You can select a menu item either with the keyboard or the mouse. Windows then sends a message indicating which item was selected.

### Menu definition in resource scripts

Since each menu must have a specific name, more than one menu can be defined in the resource script simultaneously.



The following menu definition is part of a resource script (you'll find the source code and this resource script on the companion diskette under the name FRSTMENU in the WINPRGDE\RESOURCE\FRSTMENU directory):

```
FirstMenu MENU
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "&New",           IDM_NEW
        MENUITEM "&Open",          IDM_OPEN
        MENUITEM "&Save",          IDM_SAVE
        MENUITEM "Save &As",      IDM_SAVEAS
        MENUITEM SEPARATOR
        MENUITEM "E&xit",         IDM_EXIT
        MENUITEM "A&bout",        IDM_ABOUT
    END
    POPUP "&Edit"
    BEGIN
        MENUITEM "&Undo",          IDM_UNDO
        MENUITEM "Cu&t",          IDM_CUT, GRAYED
        MENUITEM "&Copy",          IDM_COPY, GRAYED
        MENUITEM "&Paste",         IDM_PASTE
    END
    MENUITEM "\a&Help",         IDM_HELP
END
```

The menu name, which is the connection to the source text, precedes the MENU data type. A menu can consist of various pop-up menus and menu commands which are kept together by BEGIN and END statements.

A POPUP statement defines a pop-up menu, which in turn contains a pop-up menu and/or menu entries. When the user selects a pop-up menu, Windows displays the list of items that belong to it.

A MENUITEM statement defines a menu item with its ID value. You can use #define to assign a symbolic name to the ID value. However, this name should be as specific as possible because it will be used in the window function.

Instead of displaying text, a menu entry can also display a bitmap. You can add options to the menu entries as well as the pop-up menus. In this example, the menu items Cut and Copy aren't executable.

These menu items are inactive because they contain the GRAYED option. With this option, the menu items appear in a different color than the menu text color. An entry that has the INACTIVE option also cannot be selected although it looks like an active entry.

Another option that is often used is CHECKED. This option places a mark (check mark) in front of the menu item or pop-up menu.

You can link several options by using the bitwise OR operation. The MENUITEM SEPARATOR line creates an inactive item that's used as a dividing bar for two other entries. If this statement is used in the menu line, the separator is vertical.

Certain control characters can also be used within a text string. For example, if an ampersand (&) is placed in front of a character in a menu item, this character is underlined when displayed. To select the menu item in which this character appears, press **Alt** and the underlined character.

The control character \a should only be used for pop-up menus or menu items in the menu bar because it causes the item to be displayed to the right of the line. Otherwise, Windows fills the menu bar from left to right and displays it in several lines if it runs out of space.

## Linking to the source text

After defining the menu in the .RC file, you must link the menu to the source text. Main windows and pop-up windows can have menus but child windows cannot.

There are two methods to link a menu to the source text. One method is to specify the name of the menu as a string in the WNDCLASS structure. Then all windows that refer to this class will have the same menu:

```
wcMenuClass.lpszMenuName = "FirstMenu";
```

Another method is to name the menu while creating a window. By doing this, windows of the same class can have different menus. The CreateWindow function requires a handle to the loaded menu. To obtain this handle, use the LoadMenu function:

```
CreateWindow("FirstMenu",
             "FirstMenu",
             WS_OVERLAPPEDWINDOW,
             CW_USEDEFAULT, 0,
             CW_USEDEFAULT, 0,
             NULL,
             LoadMenu( hInstance, "FirstMenu" ),
             hInstance,
             NULL);
```

If you specify menus both when registering a window class and creating a window, the special window menu has priority.

## Evoked messages

When the user selects an active menu with the keyboard or the mouse, a WM\_COMMAND message occurs. This message is used to process the menu items one by one because the ID value of the selected menu item is given in the wParam parameter. Frequently, a switch statement is used for wParam.

```
case WM_COMMAND:
    switch( wParam )
    {
```

```
case IDM_NEW:  
    /* User-defined data */  
    break;  
case IDM_OPEN:  
    /* User-defined data */  
    break;  
  
.  
. .  
  
default:  
    break;  
}  
break;
```

However, when you click a pop-up menu, such as **File**, a WM\_INITPOP-UP message is sent to the window function. Since a pop-up menu doesn't have an ID value, the wParam parameter contains the handle to the pop-up menu. This message is also processed when single menu items are supposed to be initialized in the pop-up menu. Otherwise, the message is passed to the default window procedure.

Shortly before the menu bar appears in the window, Windows sends the WM\_INITMENU message. In this case, wParam contains the handle to the main menu, which was named in the FirstMenu example.

You don't have to worry about pulling down the pop-up menu or removing it later because Windows handles this.

You can also add your own items to the system menu. As soon as a menu item from the system box is selected, Windows sends a WM\_SYSCOMMAND message.

All cases that you don't process yourself should be passed to DefWindowProc. Otherwise the window cannot be enlarged or closed.

## Menu changes

Ordinarily, a menu is a dynamic creation instead of a static one. This means that a menu is always changing. While an application is running, menu items can change their status, and be added or deleted, etc. Also, an entire menu can be exchanged.

You can use various functions to change menus. The following are the most important functions:

## **Changes to menu item's status:**

CheckMenuItem	Sets or removes a checkmark
EnableMenuItem	Enables, disables or "greys" a menu item
HiliteMenuItem	Sets or removes highlighting of an item in the menu bar

## **Changes to menu item's presence:**

AppendMenu	Adds a menu item
DeleteMenu	Deletes an item and its pop-up menu
InsertMenu	Inserts a menu item
ModifyMenu	Changes a menu item
RemoveMenu	Removes a menu item but does not delete it

## **Changes to the entire menu:**

CreateMenu	Creates an empty menu
DestroyMenu	Deletes a menu
SetMenu	Sets a new menu

The next example application demonstrates most of these functions through practical examples.

There are also several functions, beginning with Get, that retrieve current data. This could be the status of a menu item or a handle to the menu or a pop-up menu.

## **Information function:**

GetMenu	Obtains a menu handle
GetMenuCheckMarkDimensions	Supplies dimensions of the default checkmark
GetMenuItemCount	Supplies the number of items in a menu
GetMenuItemID	Provides the ID value of an item
GetMenuState	Passes the status of an item

GetMenuItemString	Copies the string of the item to the buffer
GetSubMenu	Obtains the handle of the pop-up menu
GetSystemMenu	Provides access to the control menu

A menu item can also consist of a bitmap instead of a text string. However, this can be specified only during runtime, not in the .RC file.

You either add or insert a new entry or modify an existing entry. For all three functions, you must pass the MF\_BITMAP flag and the handle of the bitmap.

## Floating pop-up menus

Instead of appearing underneath the title bar, this type of menu can appear anywhere on the screen. For example, the pop-up menu can appear wherever the pointer is in the client area when you press the right mouse button.

Both the CreatePopupMenu and TrackPopupMenu functions are needed to create a floating pop-up menu. The first function creates an empty pop-up menu and retrieves its handle. Then you can use the AppendMenu function to fill this empty menu with menu items.

Before the floating pop-up menu can be displayed on the screen, you must convert the coordinates of the menu to screen coordinates. You can determine the coordinates of the menu from the lParam parameter of the WM\_RBUTTONDOWN message. After the user selects an item from this menu, the screen automatically removes the menu (see the following sample application).

## Defining your own checkmarks

Instead of using the predefined checkmarks, it's possible to create your own bitmap to use as a checkmark. You can also use a bitmap to show that an item isn't selected.

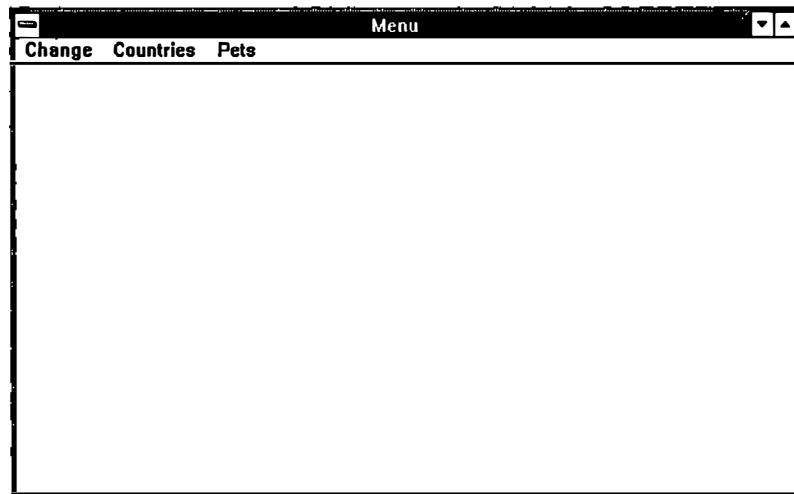
First the bitmaps must be painted, for example, with SDKPAINT.EXE. Remember that your own checkmarks should be the same size as the predefined checkmarks.

The bitmaps are specified as single-line statements in the .RC file. Use LoadBitmap to load each bitmap in source code. Then use SetMenuItemBitmaps to assign one bitmap for MF\_CHECKED status and a different one for MF\_UNCHECKED status.

Frequently you also must adjust the size of the drawn bitmap according to the dimensions of the predefined checkmark. Use the GetMenuCheckMarkDimensions function to determine the dimensions of the predefined checkmark. Then adjust the bitmap size with the StretchBlt function. If the size of the bitmap is only slightly smaller than the default, you may not have to adjust its size (see the following sample application).

You can also create custom menu items (e.g., ones that are much larger than usual). Assign the MF\_OWNERDRAW style to these entries. However, in these instances the programmer, instead of Windows, is responsible for drawing the menu item. The first time the menu is drawn, the window function receives the WM\_MEASUREITEM message. The lParam parameter points to a data structure where you specify the dimensions of the menu item yourself. Whenever an entry must be redrawn, Windows sends the WM\_DRAWITEM message.

## Menu example



The following examples demonstrate many of the items we discussed in this section.

New messages	Brief description
WM_COMMAND	Sent when an active menu item is clicked
New functions	Brief description
AppendMenu	Adds an item to the menu
CheckMenuItem	Sets or deletes the checkmark
ClientToScreen	Converts client coordinates to screen coordinates
CreatePopupMenu	Creates an empty pop-up menu
DeleteMenu	Deletes a menu item
DestroyMenu	Deletes the specified menu
EnableMenuItem	Changes the active status of the item
GetMenu	Obtains a handle to a menu
GetMenuCheckMarkDimensions	Supplies the size of the default checkmark
GetMenuItemString	Copies a menu text to a string
GetSubMenu	Obtains a handle to a pop-up menu
InsertMenu	Inserts a menu item
LoadBitmap	Supplies a handle to a bitmap
LoadMenu	Supplies a handle to a menu
LoadString	Loads a text from the STRINGTABLE
ModifyMenu	Changes existing menu item
SetMenu	Sets a new menu
SetMenuItemBitmaps	Sets your own bitmaps as checkmarks
TrackPopupMenu	Displays a floating pop-up menu

## Source code: MENU.C

```
/** MENU.C ****
/** Demonstration of menus and checkmarks.
 */
#include "windows.h"                                // Include windows.h header file
#include "menu.h"                                    // Include menu.h header file

BOOL MenuInit ( HANDLE );
long FAR PASCAL MenuWndProc( HWND, unsigned, WORD, LONG);

/** WinMain (main function for every Windows application ****
int PASCAL WinMain( hInstance, hPrevInstance, lpszCmdLine,cmdShow)
HANDLE hInstance, hPrevInstance;                  // Current & previous instance
```

## Resources

---

```
LPSTR lpszCmdLine;                                // Long ptr to string after
                                                    // program name during execution
int cmdShow;                                       // Specifies the application
                                                    // window's appearance
{
    MSG msg;                                         // Message variable
    HWND hWnd;                                        // Window handle

    if (!hPrevInstance)                               // Initialize first instance
    {
        if (!MenuInit( hInstance ))
            return FALSE;
    }

/** Specify appearance of application's main window *****/
    hWnd = CreateWindow("Menu",
                        "Menu",
                        WS_OVERLAPPEDWINDOW,
                        CW_USEDEFAULT, 0, CW_USEDEFAULT, 0,
                        NULL,
                        NULL,
                        hInstance,
                        NULL);

    ShowWindow( hWnd, cmdShow );                      // Make window visible
    UpdateWindow( hWnd );                            // Update window

    while (GetMessage(&msg, NULL, 0, 0)) // Message reading
    {
        TranslateMessage(&msg);                     // Message translation
        DispatchMessage(&msg);                      // Send message to Windows
    }
    return (int)msg.wParam;                          // Return wParam of last message
}

BOOL MenuInit( hInstance )                         // Menu initialization
HANDLE hInstance;
{

/** Specify window class *****/
    WNDCLASS wcMenuClass;

    wcMenuClass.hCursor      = LoadCursor( NULL, IDC_ARROW );
    wcMenuClass.hIcon        = LoadIcon( NULL, IDI_APPLICATION );
    wcMenuClass.lpszMenuName = (LPSTR) "Menul";
    wcMenuClass.lpszClassName = (LPSTR) "Menu";
```

```
wcMenuClass.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
wcMenuClass.hInstance = hInstance;
wcMenuClass.style = CS_VREDRAW | CS_HREDRAW;
wcMenuClass.lpfnWndProc = MenuWndProc;
wcMenuClass.cbClsExtra = 0 ;
wcMenuClass.cbWndExtra = 0 ;

if (!RegisterClass( &wcMenuClass ) )
    return FALSE;
return TRUE;
}

/** MenuWndProc *****/
/** Main window function: All messages are sent to this window ***/
/** *****/

long FAR PASCAL MenuWndProc( hWnd, message, wParam, lParam )
HWND      hWnd;
unsigned message;
WORD      wParam;
LONG      lParam;
{
    static HMENU    hMenu1,hMenu2, hSubMenu; // Menu handle variables
    static HMENU    hPopup = 0;
    static HANDLE   hInst;                  // Instance handle
    char          szNewItem[MAX_STRING_LENGTH];
    char          chMenuText[MAX_STRING_LENGTH];
    char          chText[20] ;
    POINT        pt;
    static BOOL    bMexico = TRUE;
    static HBITMAP  hBitCheck1, hBitNotCheck1, hBitCheck2, hBitNotCheck2;
    DWORD       dDim;
    static WORD    wHeight, wWidth;
    static int     i = 60;

    switch (message)
    {
        case WM_CREATE:
            hInst = GetWindowWord (hWnd, GWW_HINSTANCE);
            hMenu1 = GetMenu(hWnd);
            hMenu2 = LoadMenu(hInst, "Menu2");
            hSubMenu = GetSubMenu(hMenu1, 2);

            hBitCheck1 = LoadBitmap(hInst, "BitCheck1");
            hBitNotCheck1 = LoadBitmap(hInst, "BitNotCheck1");

            hBitCheck2 = LoadBitmap(hInst, "BitCheck2");
            hBitNotCheck2 = LoadBitmap(hInst, "BitNotCheck2");
    }
}
```

```
/* Optional:           dDim = GetMenuCheckMarkDimensions();
 Displays checkmark   wHeight = HIWORD(dDim);
 dimensions in a     wWidth = LOWORD(dDim);
 message box          wsprintf(chText, "x = %d, y = %d", wWidth, wHeight);
                      MessageBox(hWnd, chText, "Title", MB_OK);

*/
SetMenuItemBitmaps(hMenu1, ID_AUST, MF_BYCOMMAND,
                   hBitNotCheck1,
                   hBitCheck1);
SetMenuItemBitmaps(hMenu1, ID_GRMN, MF_BYCOMMAND,
                   hBitNotCheck2,
                   hBitCheck2);
break;

case WM_COMMAND:           // Get messages from menu bar
    switch(wParam)
    {
        case ID_INSERT: // Insert item
            LoadString( hInst, IDS_TIG, (LPSTR)szNewItem,
                         MAX_STRING_LENGTH);
            InsertMenu( hSubMenu, ID_KOAL, MF_BYCOMMAND, IDS_TIG,
                         (LPSTR)szNewItem);
            EnableMenuItem( hMenu1, ID_INSERT, MF_GRAYED);
            EnableMenuItem( hMenu1, ID_DELETE, MF_ENABLED);
            break;

        case ID_DELETE: // Delete item
            EnableMenuItem( hMenu1, ID_DELETE,
                            MF_GRAYED);
            EnableMenuItem( hMenu1, ID_INSERT,
                            MF_ENABLED);
            DeleteMenu( hSubMenu, 3, MF_BYPOSITION);
            break;

        case ID_APPEND: // Append item
            LoadString( hInst, IDS_LIN,
                         (LPSTR)szNewItem,
                         MAX_STRING_LENGTH);
            AppendMenu(hSubMenu, MF_ENABLED, IDS_LIN,
                         (LPSTR)szNewItem);
            break;

        case ID MODIFY: // Modify item
            LoadString( hInst, IDS_CHE,
                         (LPSTR)szNewItem,
                         MAX_STRING_LENGTH);
            ModifyMenu( hMenu1, ID_GIRA, MF_BYCOMMAND,
```

```
    IDS_CHE,
    (LPSTR)szNewItem);
EnableMenuItem( hMenu1, ID MODIFY,
                MF_GRAYED);
break;

case ID_POPCRE: // Create Pop-up item
    hPopup = CreatePopupMenu();
    EnableMenuItem( hMenu1, ID_POPDES,
                    MF_ENABLED);
    EnableMenuItem( hMenu1, ID_POPCRE,
                    MF_GRAYED);
break;

case ID_POPDES: // Destroy Pop-up item
    DestroyMenu( hPopup);
    hPopup = 0;
    EnableMenuItem( hMenu1, ID_POPCRE,
                    MF_ENABLED);
    EnableMenuItem( hMenu1, ID_POPDES,
                    MF_GRAYED);
break;

case ID_MENU1: // Change Menu item (toggle)
    SetMenu(hWnd, hMenu2);
break;
case ID_MENU2: // Change Menu item (toggle)
    SetMenu(hWnd, hMenu1);
break;

case ID_GRMN: // Germany item
    CheckMenuItem(hMenu1, ID_AUST,
                  MF_BYCOMMAND |
                  MF_UNCHECKED);
    CheckMenuItem(hMenu1, ID_GRMN,
                  MF_BYCOMMAND |
                  MF_CHECKED);
break;

case ID_AUST: // Australia item
    CheckMenuItem(hMenu1, ID_AUST,
                  MF_BYCOMMAND | MF_CHECKED);
    CheckMenuItem(hMenu1, ID_GRMN,
                  MF_BYCOMMAND |
                  MF_UNCHECKED);
break;

case ID_MEXI: // Mexico item
```

```
        if (bMexico)
        {
            CheckMenuItem(hMenu1, ID_MEXI,
                          MF_BYCOMMAND |
                          MF_UNCHECKED);
            bMexico = FALSE;
        }
        else
        {
            CheckMenuItem(hMenu1, ID_MEXI,
                          MF_BYCOMMAND |
                          MF_CHECKED);
            bMexico = TRUE;
        }
        break;

    case ID_CATT:      // Cat item
    case ID_DACH:      // Dachshund item
    case ID_POOD:      // Poodle item
    case ID_SETT:      // Setter item
    case ID_TERR:      // Terrier item
    case ID_GPIG:      // Guinea Pig item
    case ID_PRKT:      // Parakeet item
        if (hPopup != 0)
        {
            GetMenuItemString(hMenu1, wParam,
                               (LPSTR)chMenuText,
                               MAX_STRING_LENGTH,
                               MF_BYCOMMAND);
            AppendMenu(hPopup, MF_ENABLED, i++,
                        (LPSTR)chMenuText);
        }
        break;

    default:
        return (DefWindowProc(hWnd, message,
                             wParam,lParam));
        break;
    }
    break;

case WM_RBUTTONDOWN:           // Right button pressed?

    if (hPopup)
    {
        pt = MAKEPOINT( lParam);
        ClientToScreen (hWnd, (LPOINT)&pt);
```

```

        TrackPopupMenu( hPopup, 0, pt.x, pt.y, 0,
                        hWnd,NULL);
    }
    break;

    case WM_DESTROY:           // Destroy window
        PostQuitMessage(0);
        break;

    default:
        return (DefWindowProc( hWnd, message, wParam, lParam ));
    }
    break;
}
return(0L);
}

```

## Module definition file: MENU.DEF

```

NAME      Menu

DESCRIPTION 'Menu example'

EXETYPE   WINDOWS

STUB      'WINSTUB.EXE'

CODE      PRELOAD MOVEABLE
DATA      PRELOAD MOVEABLE MULTIPLE

HEAPSIZE  4096
STACKSIZE 4096

EXPORTS   MenuWndProc  @1

```

## Resource script: MENU.RC

```

/** MENU.RC ****
/** Resource file for MENU.C. Defines bitmaps and menu information. ***/
/** ****

#include "menu.h"                                // Include menu.h header file
#include "windows.h"                             // Include windows.h header file

BitCheck1     BITMAP  CHECK1.bmp
BitNotCheck1  BITMAP  NOCHECK1.BMP

```

## *Resources*

---

```
BitCheck2      BITMAP  CHECK2.bmp
BitNotCheck2   BITMAP  NOCHECK2.BMP

Menu1  MENU
BEGIN
    POPUP  "Change"
        BEGIN
            MENUITEM "Insert",           ID_INSERT
            MENUITEM "Delete",          ID_DELETE, GRAYED
            MENUITEM "Append",          ID_APPEND
            MENUITEM "Modify",          ID MODIFY
            MENUITEM "Create Pop-up",   ID_POPCRE
            MENUITEM "Destroy Pop-up",  ID_POPDES, GRAYED
            MENUITEM "Change Menus",    ID_MENU1,
        END

    POPUP  "Countries"
        BEGIN
            MENUITEM "Australia",     ID_AUST
            MENUITEM "Germany",       ID_GRMN
            MENUITEM "Mexico",        ID_MEXI,   CHECKED
        END

    POPUP  "Pets"
        BEGIN
            POPUP "House Pets"
                BEGIN
                    MENUITEM "Cat",           ID_CATT
                    POPUP "Dog"
                        BEGIN
                            MENUITEM "Dachshund", ID_DACH
                            MENUITEM "Poodle",     ID_POOD
                            MENUITEM "Setter",     ID_SETT
                            MENUITEM "Terrier",    ID_TERR
                        END
                    MENUITEM "Guinea Pig",   ID_GPIG
                    MENUITEM "Parakeet",     ID_PRKT
                END

            MENUITEM "Elephant",      ID_ELEP
            MENUITEM "Giraffe",       ID_GIRA
            MENUITEM "Koala",         ID_KOAL
            MENUITEM "Emu",           ID_EMU
        END
    END

Menu2  MENU
```

---

```

BEGIN
    POPUP "Change"
        BEGIN
            MENUITEM "Change Menus", ID_MENU2
        END

    POPUP "Plants"
        BEGIN
            MENUITEM "Tree", ID_TREE
            MENUITEM "Flower", ID_BLOO
            MENUITEM "Shrub", ID_SHRB
        END
END

STRINGTABLE
BEGIN
    IDS_TIG,           "Tiger"
    IDS_LIN,           "Lion"
    IDS_CHE,           "Cheetah"
END

```

## Header file: MENU.H

```

#define ID_INSERT      10
#define ID_DELETE      11
#define ID_APPEND      12
#define ID MODIFY      13

#define ID_POPCRE      14
#define ID_POPDES      15
#define ID_MENU1       16
#define ID_MENU2       17

#define ID_AUST 20
#define ID_GRMN 21
#define ID_MEXI 22

#define ID_CATT 30
#define ID_DACH 310
#define ID_POOD 311
#define ID_SETT 212
#define ID_TERR 213
#define ID_GPIG 32
#define ID_PRKT 33

#define ID_ELEP 41

```

```
#define ID_GIRA 42
#define ID_KOAL 43
#define ID_EMU 44

#define ID_TREE 51
#define ID_BLOO 52
#define ID_SHRB 53

#define MAX_STRING_LENGTH 20

#define IDS_TIG 100
#define IDS_LIN 101
#define IDS_CHE 102
```

To compile and link this application, use the RCOMPILE.BAT described earlier in this chapter. Here's the listing again:

```
cl -c -Gw -Zp %1
rc -r %1.rc
link /align:16 %1,%1.exe,,libw+slibcew,%1.def
rc %1.res
```

Save this file to the directory containing your source, module definition and resource scripts. Type the following and press **Enter** to compile MENU:

```
rcompile menu
```

## How MENU.EXE works

This application can cause two different menus, which are both defined in the .RC file, to appear alternately in the menu bar. Each menu contains the MenuChange item, which is used to display the other menu with the SetMenu function. The required handles were obtained during the WM\_CREATE message.

Since the first menu is entered in the WNDCLASS structure as a class menu, GetMenu provides its handle. However, the second menu must be loaded by LoadMenu.

The **Insert**, **Delete**, **Append**, and **Modify** menu items from the **Change** pop-up menu all refer to the **Pets** pop-up menu. When you

select the **Insert** item, the string with the IDS\_TIG ID value from the string table, which is also in the .RC file, is loaded into a buffer by the LoadString function.

Next, the string is inserted in front of the menu item with the ID\_KOAL ID value in the pop-up menu by InsertMenu. Now if you select the **Pets** menu, you'll notice the **Tiger** item. Since this new item shouldn't appear more than once, the **Insert** entry must have the GRAYED status. The **Delete** item, which was initialized in the .RC file with GRAYED, is set to executable.

If you select the **Delete** item, the new item is deleted from the **Pets** pop-up menu by DeleteMenu. The flag is set to MF\_BYPOSITION. This causes the second parameter to specify the position within the pop-up menu instead of the ID value.

The first entry has the position of 0. The handle of the pop-up menu that is needed for this was determined by the GetSubMenu function during the WM\_CREATE message. At the end, the characteristics of both the **Insert** and **Delete** entries are exchanged.

The **Append** menu item loads the text "Lion" from the string table and adds it to the end of the **Pets** pop-up menu with the AppendMenu function. The status of the new menu item is set to MF\_ENABLED and the item is given the same ID value that it had as a string ID in the string table. However, this isn't mandatory.

The ModifyMenu function changes the text and the ID value of the menu item with the ID\_GIRA identification, which was valid until this change. The **Giraffe** menu item name changes to **Cheetah**. The third, fourth, and fifth parameters contain specifications for the new item.

The last two items of the **Change** pop-up menu, **Create Pop-up** and **Destroy Pop-up**, apply to a floating pop-up menu. The **Create Pop-up** menu item creates an empty pop-up menu.

This menu is filled with entries when items from the **House Pets** pop-up submenu are selected. The GetMenuItemString function loads the menu text, set by the ID value, into a buffer that is added to the end of the menu by AppendMenu.

As soon as an empty menu is available, it can be deleted with DestroyMenu. Then the handle of the **Change** pop-up menu is set to 0. DestroyMenu releases the occupied memory area. Whenever you click **Create Pop-up or Destroy Pop-up**, you must change the status of MF\_ENABLED to MF\_GRAYED or vice versa.

When the right mouse button is pressed, the floating pop-up menu, if it currently exists, is displayed wherever the mouse cursor is located. The position of the mouse, which is provided in the lParam parameter during the WM\_RBUTTONDOWN message, can be placed in a variable of the POINT structure by the MAKEPOINT macro.

The ClientToScreen function converts this item into screen coordinates, which then apply to the upper-left corner of the screen. This is necessary for the TrackPopupMenu function, which displays the pop-up menu.

The **Countries** pop-up menu has three menu items. Each of these items has a different checkmark. After each selection, the **Mexico** entry changes its status by using the CheckMenuItem function. This is displayed with the predefined checkmark.

The other two items switch each other on and off. They do this by using self-defined checkmarks, both for the MF\_CHECKED status and MF\_UNCHECKED status.

The bitmaps for these checkmarks are loaded with LoadBitmap during the WM\_CREATE message. They were created by SDKPAINT.EXE and their dimensions are 10 x 10. In this case, it isn't necessary to adjust these dimensions according to the predefined checkmark. As an experiment, a few lines that were used are now in the comment lines. You can use these lines to display a message box with the data of the predefined checkmark. The SetMenuItemBitmaps function passes one handle of a loaded bitmap for selected status and one for nonselected status for each of the menu items.

The .RC file contains four single-line statements that apply to bitmaps, a string table, and two menus. In the first menu, the menu items **Delete** and **Destroy Pop-up** have the initialization value of GRAYED while the **Mexico** item has the value of CHECKED.

---

All items have ID values defined in a separate header file. The file is linked to the .RC file and the .C file.

## Keyboard accelerators

Accelerator keys are used to select menu items without opening the pop-up menu. This makes applications more user-friendly.

These keys belong to the resource script file. They are stored in an ACCELERATORS table, which also contains the ID value of the menu item to which the accelerator applies. There are four ways these keys can be defined.

```
AccelMenu ACCELERATORS
BEGIN
    "^\r",    ID_INSERT
    "A",      ID_APPEND
    77,       ID MODIFY
    VK_DELETE, ID_DELETE, SHIFT, ALT, VIRTKEY
END
```

The number 77 is the decimal ASCII code for "M". The keyword VIRTKEY must be written for all virtual keys (e.g., all function keys). Also, any of the appropriate menu items should inform the user that an accelerator key is available.

```
Menu1 MENU
BEGIN
    POPUP "&Change"
    BEGIN
        MENUITEM "Insert\t CONTROL+I",      ID_INSERT
        MENUITEM "Delete\tDEL+SHIFT+ALT",   ID_DELETE, GRAYED
        MENUITEM "Append\t A",             ID_APPEND
        MENUITEM "Modify\t M",            ID MODIFY
        MENUITEM "PopupCreate",          ID_POPCRE
        MENUITEM "PopupDestroy",         ID_POPDES, GRAYED
        MENUITEM "MenuChange",           ID_MENU1
    END
    .
    .
    .
END
```

So that Windows recognizes these keys as accelerator keys, you must load this table with the LoadAccelerators function. The second parameter is the name of the table (e.g., AccelMenu).

Also, an extra function must be entered in the message loop:

```
while( GetMessage( &msg, NULL, 0, 0 ) )
{
    if( !TranslateAccelerator( hWnd, hAccel, &msg ) )
    {
        TranslateMessage( &msg );
        DispatchMessage( &msg );
    }
}
```

Each message that is obtained from the application message queue with GetMessage is passed to the TranslateAccelerator function. If, with the help of the handle to the loaded accelerator table, this function notices that the message is an accelerator key, it converts the message to a WM\_COMMAND message, retrieves a return value of TRUE, and passes the WM\_COMMAND message to the window function.

Therefore, the DispatchMessage function cannot run. The identification of the accelerator, which at the same time is also the ID value of the menu item, is sent in the wParam parameter.

## Dialog boxes

A dialog box is a pop-up window that can be used by an application to exchange information with the user. Different controls, which display the contents of a dialog box, are used. Each dialog box consists of a window and its window procedure.

## Controls

A control is a child window with a class predefined in Windows and its window function. So the appearance and tasks of a particular window are already set. You can also create controls that apply to a class defined in the application. These are called custom controls. There are seven predefined classes:

## Control classes:

BUTTON	Creates a small window for clicking alternatives (yes, no, etc.)
COMBOBOX	Creates a window that is a combination of EDIT and LISTBOX
EDIT	Creates a window for editing text
LISTBOX	Creates a window from which you select a name from a list
MDICLIENT	Creates client windows, which replace the client area in MDI (Multiple Document Interface)
SCROLLBAR	Creates a window that looks and functions like a scroll bar
STATIC	Creates a window containing fixed text or simple graphics

Along with the class, there are also style parameters that influence the way a control looks and operates. For example, with button controls you use style parameters to choose one of four different kinds of buttons.

Style	Description
BS_PUSHBUTTON	Button in the shape of an ellipse
BS_CHECKBOX	Button in the shape of a rectangle
BS_RADIOBUTTON	Button in the shape of a circle
BS_GROUPBOX	Rectangle where other buttons can be grouped

A push button can also have the BS\_DEFPUSHBUTTON style parameter. To simplify programming, the BS\_AUTOCHECKBOX and BS\_AUTORADIOBUTTON styles can be used for check boxes and radio buttons.

The check box or radio button is automatically checked or unchecked, depending on its previous status, without having to send the BM\_SETCHECK message to the button. Also, a radio button that had

been set is switched off within the same group because radio buttons permit only a "1:n" selection.

Besides dialog boxes, controls also appear in normal windows. Each control can be identified by its ID, which is similar to the menu ID.

The user could process the control by typing input. The control then passes certain information to its parent window by sending a WM\_COMMAND message to the window procedure of the parent window.

The parameter wParam contains the ID of the control while lParam contains the handle of the control in the low-order word and the notification code in the high-order word. This code provides detailed information about what occurs in the control. Notification codes can be divided into four groups, based on the control from which they result.

## **Notification codes:**

Type	Example
Button NC	BN_CLICKED
Edit-Control NC	EN_SETFOCUS
Listbox NC	LBN_DBLCLK
Combobox NC	CBN_EDITCHANGE

For example, when a radio button is double-clicked in a dialog box, this button sends the WM\_COMMAND message, with the BN\_DOUBLECLICKED code, to the window function of the dialog box.

If the control is supposed to perform a certain task, the parent window uses the SendMessage function to send a control message to the child window. In order for the parent window to send the message, it must know the handle of the control because it is the first parameter.

In dialog boxes you can determine the handle by using the GetDlgItem function. You could also use the SendDlgItemMessage function, which uses only the ID value of the control.

```
hControl = GetDlgItem( hDlg, .Control-ID );
SendMessage( hControl, message, wParam, lParam );
```

or

```
SendDlgItemMessage( hDlg, Control-ID, message, wParam, lParam );
```

There are different groups of control messages that apply to a certain class and, depending on the class, begin with different letters (BM\_, EM\_, LB\_, CB\_, etc.).

For example, there is the BM\_SETCHECK message, which is sent to a radio button or check box to change the current status. If the parameter has a value that doesn't equal 0, the check mark is set. Otherwise, the checkmark is deleted. You can achieve the same result by using the CheckDlgButton or CheckRadioButton function.

```
SendDlgItemMessage( hDlg, Control-ID, BM_SETCHECK, TRUE, 0L );
```

or

```
CheckDlgButton( hDlg, Control-ID, TRUE );
```

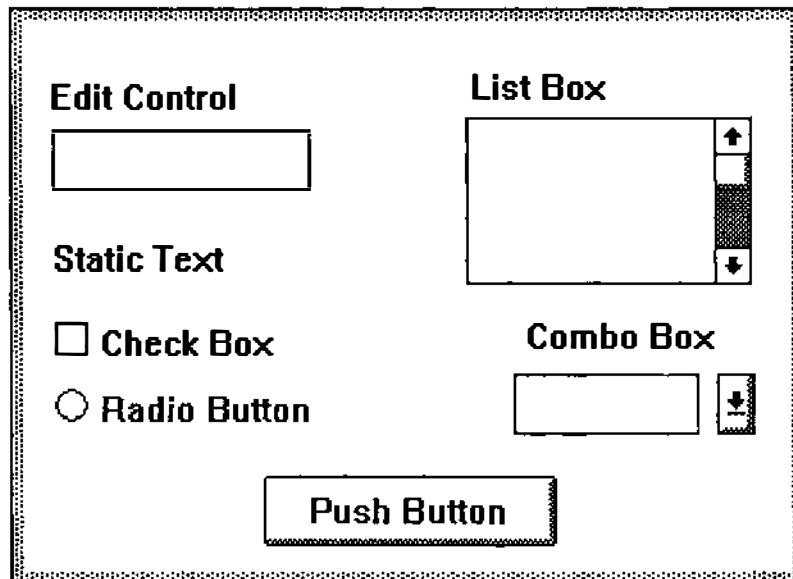
or

```
CheckRadioButton( hDlg, FirstC-ID, LastC-ID, Control-ID );
```

The CheckRadioButton function can be used only for a group of radio buttons, not a single button. The second parameter is the ID value of the first radio button of the group, and the third parameter contains the ID value of the last radio button of the group. The fourth parameter represents the button that will be set.

All the other check marks within this group are deleted. This is true even if the radio buttons don't have the BS\_AUTORADIOBUTTON style.

The following illustration shows the most frequently used controls:



## Creating dialog boxes

There are two ways to create a dialog box. One way is to specify it line by line in the .RC file. However, this is more complicated than defining a menu because each control needs information about its location within the box.

You could also use the DIALOG.EXE utility application to create the dialog box on the screen. Also, the dialog editor enables you to integrate controls with user-defined window procedures so that you can use them like default controls.

A menu called Test is located in the Dialog pop-up menu. Use this item to test a finished dialog box. For example, you can check whether all the radio buttons and check boxes were defined with the BS\_AUTORADIOBUTTON, BS\_AUTOCHECKBOX, and BS\_3STATE style parameters.

You can also determine whether the WS\_GROUP and WS\_TABSTOP styles are assigned to the proper controls. These parameters provide information about when the [Tab] and cursor keys are used to move between individual controls. Usually the cursor keys handle controls that belong to the same group and the [Tab] key handles controls that belong

to different groups. For example, a group can consist of three radio buttons enclosed by a group box.

While the dialog box is being saved, the dialog editor creates two files: an ASCII file with the .DLG extension and a file in binary format with the .RES extension. If the editor is used to make changes to the box, the second file is needed. Although you can also make changes directly in the .DLG file, these changes aren't automatically accepted in the .RES file.

First, two steps are needed in the .RC file:

```
#include "filename.dlg"
```

Then start the resource compiler to create a new .RES file from the .RC file. You can use the dialog editor to read this file, even if it contains other resources. Instead of using the #include statement, you could also copy the entire .DLG file to the .RC file. However, if you copy several dialog boxes, it's difficult to keep them organized.

A .DLG file can look as follows:

```
MYBOX DIALOG LOADONCALL MOVEABLE DISCARDABLE 46, 19, 247, 169
CAPTION "Default"
STYLE WS_BORDER | WS_CAPTION | WS_DLGFRADE | WS_POPUP
BEGIN
    CONTROL "Text Search", ID_SRCH, "button", BS_PUSHBUTTON | WS_GROUP | WS_TABSTOP
    | WS_CHILD, 11, 45, 50, 14
    CONTROL "Accept Title", ID_NEW, "button", BS_PUSHBUTTON | WS_TABSTOP | WS_CHILD,
3, 83, 68, 14
    CONTROL "New Title", -1, "static", SS_LEFT | WS_GROUP | WS_CHILD, 81, 17, 54, 8
    CONTROL "", ID_COMB, "combobox", CBS_DROPDOWN | WS_VSCROLL | WS_CHILD, 81, 28,
55, 85
    CONTROL "Lines", -1, "button", BS_GROUPBOX | WS_GROUP | WS_CHILD, 153, 15, 65,
61
    CONTROL "Left->Right", ID_LERI, "button", BS_AUTOCHECKBOX | WS_TABSTOP |
WS_CHILD, 155, 33, 58, 12
    CONTROL "Right->Left", ID_RILE, "button", BS_AUTOCHECKBOX | WS_CHILD, 155, 49,
59, 12
    CONTROL "Icon", -1, "button", BS_GROUPBOX | WS_GROUP | WS_CHILD, 151, 96, 64, 63
    CONTROL "Stop", ID_ICON, "button", BS_AUTORADIOBUTTON | WS_TABSTOP | WS_CHILD,
155, 109, 47, 12
    CONTROL "Question", ID_ICON+1, "button", BS_AUTORADIOBUTTON | WS_CHILD, 155,
125, 47, 12
```

```
CONTROL "Exclamation", ID_ICON+2, "button", BS_AUTORADIOBUTTON | WS_CHILD, 155,  
140, 58, 12  
CONTROL "OK", IDOK, "button", BS_DEFPUSHBUTTON | WS_GROUP | WS_TABSTOP |  
WS_CHILD, 18, 144, 36, 14  
CONTROL "Cancel", IDCANCEL, "button", BS_PUSHBUTTON | WS_TABSTOP | WS_CHILD, 75,  
144, 39, 14  
END
```

This dialog box "MyBox" is used in the following example. It consists of thirteen controls: a combobox, a static field, two group boxes, four push buttons, two check boxes, and three radio buttons. Each control has its own ID value.

Most of the ID values in this example are in a header file with names. Controls that cannot be used because of technical reasons can have the same ID (e.g., -1).

The ID names IDOK and IDCANCEL are defined in the WINDOWS.H header file and should be used for the **OK** and **Cancel** push buttons. You can use the **Tab** key to start controls that have the WS\_TABSTOP style.

You can also draw your own icon in the dialog box by selecting the Icon item in the dialog editor, which enters a placeholder in the .DLG file. As with many of the other controls, you can also assign text to this icon. In the .RC file, this text is then equal to the name of the icon in the ICON statement.

.DLG File:

Control "House", ID\_HOUSE, "static", SS\_ICON / WS\_CHILD, 35, 115, 18, 21

.RC File:

House" ICON house.ico

In this example, the name "House" was selected. The ID value for this control of the "static" class is ID\_HOUSE and the file containing the icon is called HOUSE.ICO.

## Types

There are two types of dialog boxes. The main difference between these boxes is the way they react to the window they activate.

modeless dialog box  
modal dialog box

When a window procedure encounters a modeless dialog box, this dialog box runs parallel to the window belonging to the window procedure without influencing this window in any way. Other applications also continue to work. Most modeless dialog boxes have the WS\_POPUP, WS\_CAPTION, WS\_BORDER, and WS\_SYSTEMMENU styles. You can close this dialog box by selecting Close in the Control menu.

However, a modal dialog box puts its parent window in standby position. So, the parent window cannot continue working until the user exits the dialog box. In addition to this application modal dialog box, there is also a system modal dialog box. This dialog box stops all other Windows programs until it is closed with **OK** or **Cancel**.

Since it interferes with Windows multitasking, you should use the system modal dialog box as little as possible. Modal dialog boxes have the WS\_POPUP and WS\_DLGFRAAME styles, which produce a frame around the dialog box.

## Dialog routine

A dialog(box) routine is a callback function that Windows calls when a message for the dialog function appears. The dialog routine is very similar to the normal window function. For instance, four parameters are passed to this routine. The first parameter is usually hDlg, which is a handle to a dialog window.

```
BOOL FAR PASCAL DialogProc( hDlg, wMsg, wParam, lParam )
HWND  hDlg;
WORD  wMsg;
WORD  wParam;
DWORD lParam;
```

The function must be defined with FAR PASCAL and specified, with the EXPORTS statement, along with the window function in the definition file.

Within the dialog routine, only the messages that are important for the dialog box are processed in a switch prompt. The unimportant messages aren't passed in the default branch to a default dialog box routine. Instead, the return value of the dialog routine does this. If this value is set to FALSE, Windows calls a default routine. If the value is set to TRUE, the message is processed.

Most dialog routines process the WM\_INITDIALOG and WM\_COMMAND messages. The WM\_INITDIALOG message is sent shortly before the dialog box appears on the screen. For example, you could pass the focus to a certain control by using the SetFocus function.

In such a case, however, the return value must be FALSE even though the message itself has been processed. If you retrieve TRUE, Windows resets the focus to the first control that has the WS\_TABSTOP style parameter.

```
BOOL FAR PASCAL DialogProc( hDlg, wMsg, wParam, lParam )
HWND      hDlg;
WORD      wMsg;
WORD      wParam;
DWORD     lParam;
{
    switch( wMsg )
    {
        case WM_INITDIALOG:
            SetFocus( ... );
            return( FALSE );
            break;
        case WM_COMMAND:
            switch( wParam )
            {
                case IDOK:
                    EndDialog( hDlg, 1 );
                    return( TRUE );
                    break;
                case IDCANCEL:
                    EndDialog( hDlg, 0 );
                    return( TRUE );
                    break;
            }
    }
}
```

```
    default:
        return( FALSE );
    }
default:
    return( FALSE );
}
```

Since almost every dialog box has at least one **OK** push button and often a **Cancel** push button, you need the WM\_COMMAND message, which occurs when you click a push button. The dialog box should disappear from the screen after one of these keys is pressed.

With a modeless dialog box, DestroyWindow is responsible for removing the dialog box from the screen. It's also responsible for normal windows. In a modal dialog box routine, the EndDialog function is called. The handle of the dialog box to be deleted is passed in the first parameter.

In the second parameter, you can enter a value that can be used as a return value of the DialogBox call after the dialog routine ends.

## Calling the dialog box

Regardless of whether you're calling a modal or modeless dialog box, first you must use the MakeProcInstance function. This is necessary because every dialog routine is a callback function. The MakeProcInstance function guarantees that the data segment of the current instance is used when the dialog routine begins.

So, all instances can work with the same dialog routine. The return value of the MakeProcInstance function is of the FARPROC data type and is referred to as a procedure instance address. When the dialog box is no longer needed, you must release this procedure instance address again, similar to the handle of a device context by using the FreeProcInstance function.

All callback functions, except for a window function, require both the MakeProcInstance function and the FreeProcInstance function.

```
fpProc = makeProcInstance( DialogProc, hInstance );
/* Call dialog box */
FreeProcInstance( fpProc );
```

Use the CreateDialog function to call a modeless dialog box. The second parameter represents the name of the dialog box that was given when it was created and the fourth parameter contains the procedure instance address.

Since Windows loads the dialog box and then returns to the normal window function, this routine and the dialog routine run parallel to one another. The messages to the dialog box are taken from the application message queue, within the loop message, with GetMessage. This is why you use the IsDialogMessage function in the loop to prompt for the window to which the message is addressed.

```
while( GetMessage( &msg, NULL, 0, 0 ) )
{
    if( (hDlg == NULL) || (!IsDialogMessage( hDlg, &msg )) )
    {
        TranslateMessage( &msg );
        DispatchMessage( &msg );
    }
}
```

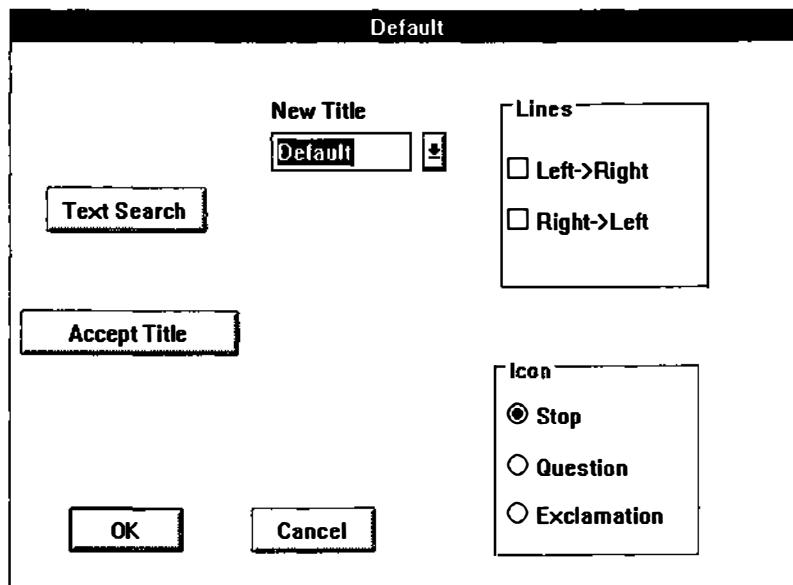
The only time the handle of the box doesn't equal NULL is when the dialog box is on the screen. This is also the only time that there can be messages to the box. The IsDialogMessage function retrieves TRUE if the message taken from the application message queue is intended for the dialog box. Then it sends this message to the dialog routine.

In this case, you can no longer call the TranslateMessage and DispatchMessage functions. The DialogBox function is needed to activate a modal dialog box. This function has the same parameters as the CreateDialog function.

```
int DialogBox ( hInstance, lpTemplateName, hWndParent, lpDialogFunc)
```

↑  
e.g "DialogboxName"  
Return value of MakeProcInstance function

## Example of a modal dialog box



In the example, we used a dialog box to make changes to the box and to the normal window.

Select an icon for the minimized version of the window from three default icons. This new setting is valid as soon as you click the **OK** push button.

Use two check boxes to decide whether no, one or two diagonal lines are drawn in the client area after the box disappears.

You can also change the title of the dialog box by using the combo box, which consists of an edit control and a list box. You can transfer the contents of this edit control to the title bar by clicking the **Accept Title** push button. During this procedure the dialog box shouldn't disappear from the screen.

The list box is filled with the filenames and subdirectories of the current directory and the existing drives during the WM\_INITDIALOG message. The user can also enter a text string in the edit control of the

combo box. This text is used as a search criterion for the contents of the list box when the **Text Search** push button is pressed.

After a successful search, the first string found is automatically displayed in the edit window. Otherwise, a message box informs the user that the text wasn't found.

#### New message

BM\_GETCHECK  
BM\_SETCHECK  
CB\_DIR  
CB\_SELECTSTRING  
WM\_COMMAND  
WM\_INITDIALOG

#### Brief description

Decides whether a button is selected  
Sets or deletes the check mark of a radio button or check box  
Inserts a filename list into the combo box  
Selects new string, part of which was specified in the edit control  
Control sends information to the parent window  
Sent shortly before the dialog box appears

#### New functions

DialogBox  
EndDialog  
FreeProcInstance  
GetDlgItem  
GetDlgItemText  
GetWindowText  
MakeProcInstance  
SendDlgItemMessage  
SetClassWord  
SetFocus  
SetWindowText

#### Brief description

Creates a modal dialog box  
Specifies the end of a modal dialog box  
Frees the procedure instance address  
Obtains handle of a control from the dialog box  
Copies text of a control to a buffer  
Copies title of a window to a buffer  
Creates procedure instance address  
Sends message to a control in a dialog box  
Changes parameter from the WNDCLASS structure  
Passes focus to a window  
Changes the title of a window

## Source code: DIALOGBX.C

```
/** DIALOGBX.C ****
/** Displays dialog box with options for specifying text and icons      */
/** ****

#include "dialogbx.h"                                     // Include dialogbx.h header file
#include "windows.h"                                      // Include windows.h header file

WORD    widIcon = ID_ICON;
BOOL    bRILE = FALSE;
BOOL    bLERI = FALSE;
```

```
BOOL FAR PASCAL DialogProc ( HWND, unsigned, WORD, LONG);
BOOL DialogbxInit ( HANDLE );
long FAR PASCAL DialogbxWndProc( HWND, unsigned, WORD, LONG);

int PASCAL WinMain(hInstance, hPrevInstance, lpszCmdLine, cmdShow)
HANDLE hInstance, hPrevInstance;           // Current & previous instance
LPSTR lpszCmdLine;                      // Long ptr to string after
                                         // program name during execution
int cmdShow;                            // Specifies the application
                                         // window's appearance
{
    MSG msg;                           // Message variable
    HWND hWnd;                         // Window handle

    if (!hPrevInstance)                // Initialize first instance
    {
        if (!DialogbxInit( hInstance ))
            return FALSE;
    }

/** Specify appearance of application's main window *****/
    hWnd = CreateWindow("Dialogbx",
                        "Dialogbx",
                        WS_OVERLAPPEDWINDOW,
                        CW_USEDEFAULT, 0,
                        CW_USEDEFAULT, 0,
                        NULL,
                        NULL,
                        hInstance,
                        NULL);

    ShowWindow( hWnd, cmdShow );         // Make window visible
    UpdateWindow( hWnd );              // Update window

    while (GetMessage(&msg, NULL, 0, 0)) // Message reading
    {
        TranslateMessage(&msg);          // Message translation
        DispatchMessage(&msg);          // Send message to Windows
    }

    return (int)msg.wParam;             // Return wParam of last message
}

BOOL DialogbxInit( hInstance )
HANDLE hInstance;
```

## Resources

---

```
/** Specify window class *****/
{
    WNDCLASS      wcDialogbxClass;

    wcDialogbxClass.hCursor      = LoadCursor( NULL, IDC_ARROW );
    wcDialogbxClass.hIcon        = LoadIcon( NULL, IDI_HAND );
    wcDialogbxClass.lpszMenuName = "Diamenu";
    wcDialogbxClass.lpszClassName = "Dialogbx";
    wcDialogbxClass.hbrBackground = GetStockObject(WHITE_BRUSH );
    wcDialogbxClass.hInstance     = hInstance;
    wcDialogbxClass.style        = CS_VREDRAW | CS_HREDRAW;
    wcDialogbxClass.lpfnWndProc   = DialogbxWndProc;
    wcDialogbxClass.cbClsExtra    = 0;
    wcDialogbxClass.cbWndExtra    = 0;

    if (!RegisterClass( &wcDialogbxClass ) )
        return FALSE;

    return TRUE;
}

/** DialogbxWndProc *****/
/** Main window function: All messages are sent to this window ***/
/********************************************/

long FAR PASCAL DialogbxWndProc( hWnd, message, wParam, lParam )
HWND      hWnd;
unsigned message;
WORD      wParam;
LONG      lParam;
{
    static HANDLE      hInst;
    FARPROC           lpprocDia;
    HICON              hIcon;
    static LPSTR        wStdIcon[3] = {IDI_HAND, IDI_QUESTION, IDI_EXCLAMATION};
    PAINTSTRUCT        ps;
    RECT               rect;
    HDC                hDC;

    switch (message)
    {
        case WM_CREATE:          // Create window
            hInst = GetWindowWord(hWnd, GWW_HINSTANCE);
            break;

        case WM_COMMAND:         // Messages from menu bar
            switch (wParam)
```

```
{  
    case ID_DIABOX:  
        lpprocDia = MakeProcInstance (DialogProc,  
                                      hInst);  
        if (DialogBox(hInst, "MyBox", hWnd,  
                      lpprocDia))  
        {  
            hIcon = LoadIcon( NULL,  
                            wStdIcon[widIcon -  
                                      ID_ICON]);  
            SetClassWord( hWnd, GCW_HICON,  
                          hIcon);  
  
            InvalidateRect(hWnd, NULL, TRUE);  
        }  
        FreeProcInstance (lpprocDia);  
        break;  
  
    default:  
        break;  
}  
break;  
  
case WM_PAINT:           // Client area redraw  
    hDC = BeginPaint(hWnd, &ps);  
    GetClientRect(hWnd, &rect);  
  
    if (bLERI)  
    {  
        MoveTo (hDC, rect.left, rect.top);  
        LineTo (hDC, rect.right, rect.bottom);  
    }  
  
    if (bRILE)  
    {  
        MoveTo (hDC, rect.right, rect.top);  
        LineTo (hDC, rect.left, rect.bottom);  
    }  
  
    EndPaint(hWnd, &ps);  
    break;  
  
case WM_DESTROY:          // Destroy window  
    PostQuitMessage(0);  
    break;
```

## Resources

---

```
default:
    return (DefWindowProc( hWnd, message, wParam, lParam ));
break;
}
return(0L);
}

BOOL FAR PASCAL DialogProc( hDlg, message, wParam, lParam )
HWND      hDlg;
unsigned    message;
WORD       wParam;
LONG       lParam;
{
    static HWND      hRadioB, hComb;          // Radio button, combo box
    WORD        wReturn;
    static char     szCombText[MAX_STRING_LENGTH];

    switch (message)
    {
        case WM_INITDIALOG:           // Dialog box initialization
            hRadioB = GetDlgItem(hDlg, widIcon);
            SendMessage(hRadioB, BM_SETCHECK, TRUE, (LONG) 0);

            SendDlgItemMessage(hDlg, ID_RILE, BM_SETCHECK, bRILE,
                                (LONG) 0);
            SendDlgItemMessage(hDlg, ID_LERI, BM_SETCHECK, bLERI,
                                (LONG) 0);

            GetWindowText(hDlg, szCombText, MAX_STRING_LENGTH);
            hComb = GetDlgItem(hDlg, ID_COMB);
            SetWindowText(hComb, szCombText);

            SetFocus(hComb);
            SendMessage(hComb, CB_DIR, 0x10 | 0x4000,
                        (LONG) (LPSTR) "*.*");
            return(FALSE);
            break;

        case WM_COMMAND:             // Receive messages
            switch (wParam)
            {
                case IDOK:   // OK button
                    bRILE = SendDlgItemMessage(hDlg, ID_RILE,
                                              BM_GETCHECK,
                                              0, 0L);
                    bLERI = SendDlgItemMessage(hDlg, ID_LERI,
                                              BM_GETCHECK,
```

```
    0, 0L);
EndDialog (hDlg, 1);
return(TRUE);
break;

case IDCANCEL: // Cancel button
    EndDialog (hDlg, 0);
return(TRUE);
break;

case ID_NEW: // Accept Title button
    GetDlgItemText( hDlg, ID_COMB, szCombText,
                    MAX_STRING_LENGTH);
    SetWindowText( hDlg, szCombText);
    return (TRUE);
break;

case ID_SRCH:// Text Search button
    GetDlgItemText(hDlg, ID_COMB, szCombText,
                   MAX_STRING_LENGTH);
    wReturn = (WORD)SendMessage(hComb,
                                CB_SELECTSTRING,
                                -1,
                                (LONG)
                                (LPSTR)
                                szCombText);
    if (wReturn == CB_ERR)
        MessageBox(hDlg,
                   (LPSTR)"No string found",
                   (LPSTR)"Error", MB_OK);
    return (TRUE);
break;

case ID_ICON:      // Stop icon
case ID_ICON+1:    // Question mark icon
case ID_ICON+2:    // Exclamation point icon
    widIcon = wParam;
    return (TRUE);
break;

default:
    return(FALSE);
}
default:
    return(FALSE);
}

}
```

## **Module definition file: DIALOGBX.DEF**

```
NAME          Dialogbx
DESCRIPTION   'Dialog box demonstration'
EXETYPE      WINDOWS
STUB          'WINSTUB.EXE'
CODE          PRELOAD MOVEABLE
DATA          PRELOAD MOVEABLE MULTIPLE
HEAPSIZE     4096
STACKSIZE    4096
EXPORTS       DialogbxWndProc @1
              DialogProc      @2
```

## **Resource script: DIALOGBX.RC**

```
#define MAX_STRING_LENGTH      20
#define ID_DIABOX               10
#define ID_SRCH                  20
#define ID_ICON                  30
#define ID_NEW                   40
#define ID_COMB                  50
#define ID_LERI                  60
#define ID_RILE                  61
```

## **Header file: DIALOGBX.H**

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

#include      "dialogbx.dlg"

Diamenu MENU
BEGIN
    MENUITEM    "Dialog Box",     ID_DIABOX
END
```

## Dialog box file: DIALOGBX.DLG

```

MYBOX DIALOG LOADONCALL MOVEABLE DISCARDABLE 46, 19, 247, 169
CAPTION "Default"
STYLE WS_BORDER | WS_CAPTION | WS_DLGFRADE | WS_POPUP
BEGIN
    CONTROL "Text Search", ID_SRCH, "button", BS_PUSHBUTTON | WS_GROUP | WS_TABSTOP
                                | WS_CHILD, 11, 45, 50, 14
    CONTROL "Accept Title", ID_NEW, "button", BS_PUSHBUTTON | WS_TABSTOP
                                | WS_CHILD, 3, 83, 68, 14
    CONTROL "New Title", -1, "static", SS_LEFT | WS_GROUP | WS_CHILD, 81, 17, 54, 8
    CONTROL "", ID_COMB, "combobox", CBS_DROPDOWN | WS_VSCROLL
                                | WS_CHILD, 81, 28, 55, 85
    CONTROL "Lines", -1, "button", BS_GROUPBOX | WS_GROUP
                                | WS_CHILD, 153, 15, 65, 61
    CONTROL "Left->Right", ID_LERI, "button", BS_AUTOCHECKBOX | WS_TABSTOP
                                | WS_CHILD, 155, 33, 58, 12
    CONTROL "Right->Left", ID_RILE, "button", BS_AUTOCHECKBOX
                                | WS_CHILD, 155, 49, 59, 12
    CONTROL "Icon", -1, "button", BS_GROUPBOX | WS_GROUP | WS_CHILD, 151, 96, 64, 63
    CONTROL "Stop", ID_ICON, "button", BS_AUTORADIOBUTTON | WS_TABSTOP
                                | WS_CHILD, 155, 109, 47, 12
    CONTROL "Question", ID_ICON+1, "button", BS_AUTORADIOBUTTON
                                | WS_CHILD, 155, 125, 47, 12
    CONTROL "Exclamation", ID_ICON+2, "button", BS_AUTORADIOBUTTON
                                | WS_CHILD, 155, 140, 58, 12
    CONTROL "OK", IDOK, "button", BS_DEFPUSHBUTTON | WS_GROUP | WS_TABSTOP
                                | WS_CHILD, 18, 144, 36, 14
    CONTROL "Cancel", IDCANCEL, "button", BS_PUSHBUTTON | WS_TABSTOP
                                | WS_CHILD, 75, 144, 39, 14
END

```

To compile and link this application, use the RCOMPILE.BAT batch file described earlier in this chapter. Here's the listing again:

```

cl -c -Gw -Zp %1
rc -r %1.rc
link /align:16 %1,%1.exe,,libw+slibcew,%1.def
rc %1.res

```

Save this file to a directory contained in your path or one containing the source, module definition and resource scripts. Type the following and press **[Enter]** to compile DIALOGBX:

```
rcompile dialogbx
```

## How DIALOGBX.EXE works

Let's examine what this application does, then we'll discuss the whys and wherefores. When you run the DIALOGBX application, it displays a window and a single menu title: Dialog Box. Clicking on this menu title displays a dialog box. This dialog box has a title bar named Default, a series of push buttons, a list box, two check boxes and three radio buttons. The Left->Right and Right->Left check boxes specify whether the application should draw diagonal lines in the main application window.

The New Title list box lists filenames. Select a filename and click the **Text Search** button. If this filename exists, nothing will happen. If you enter a nonexistent filename and click the **Text Search** button, a message box appears, indicating an error. If you select a filename and click on the **Accept Title** button, the dialog box title changes to the selected filename.

The three Icon radio buttons specify the appearance of the minimized icon. The STOP, QUESTION and EXCLAMATION icons are standard Windows icons. Clicking the **OK** button exits and implements these changes. If you minimize the main application's window, the icon used will be the icon you selected in the dialog box.

The dialog box should be called when the **Dialog Box** menu title is selected. This is why the WM\_COMMAND message prompts for the wParam parameter to the menu ID ID\_DIABOX in the window procedure. Before the dialog box can be displayed, you need a FARPROC pointer to the dialog box function. You can obtain this pointer from the MakeProcInstance function. After you select **Dialog Box**, Windows automatically starts the DialogbxWndProc window procedure. You should set a new icon and draw one or two diagonal lines only if the dialog box is closed with **OK** (i.e., if the return value is 1).

To use a new icon for the window, first you must load the icon. The second parameter of the LoadIcon function is the ID value of the desired default icon, which is taken from an array. This array contains the three options that can be selected in the dialog box. Then the handle of the

icon is passed to the SetClassWord function. You can use this function to change some fields of the WNDCLASS structure according to registration of class. Since, in this case, the icon field is supposed to be reset, we specify the handle of the new icon in the second parameter using the GCW\_HICON keyword.

Also, the InvalidateRect function places a WM\_PAINT message in the message queue. When this message is being processed, both Boolean variables can be checked for their status. If the status is TRUE, a diagonal line is drawn in the client area, either from the upper-left corner to the lower-right corner or else from the upper-right corner to the lower-left corner. The GetClientRect function obtains the dimensions of the client area.

Regardless of the return value of the DialogBox function, the pointer to the dialog procedure must be released with FreeProcInstance.

The DialogProc dialog procedure processes the messages WM\_INITDIALOG and WM\_COMMAND. All other values are passed to an internal default procedure by the FALSE return value.

In the WM\_INITDIALOG message, the radio buttons and check boxes are set to the status set in the last dialog box you closed with **OK**. The BM\_SETCHECK message sets one of the buttons when the wParam parameter contains the value TRUE. The SendDlgItemMessage function sends this message to the two check boxes. The other option is used for the radio button. The GetDlgItem function obtains the handle of the control, which is then passed to the SendMessage function.

SetFocus also passes the focus to the edit control of the combo box. So, this message must be closed with return (FALSE). The GetDlgItem function provides the handle of the combo box. The contents of the dialog box title should appear immediately in the edit control. You can use GetWindowText to write the title to a buffer and then use SetWindowText to place the title in the edit control of the combo box. The first function requires the handle of the dialog box while the second function requires the handle of the combo box.

The list box of the combo box should be filled when the dialog box is displayed. Send the CB\_DIR message to the combo box to fill the list box. The wParam parameter contains DOS file attributes that you can get from the description of the DlgDirList function. Along with

filenames, any subdirectories and the existing disk drives should be included in the display. The lParam parameter is a pointer to the string that specifies the files for the search. In this example we specified the wildcard combination "\*.\*".

In the WM\_COMMAND message, the ID value of the control (which caused this message) is passed in wParam. If you press the **Accept Title** push button, the contents of the dialog box title must be changed. The GetDlgItemText function writes the text in the edit control of the combo box to a buffer. The SetWindowText function, which we used in the WM\_INITDIALOG message, transfers this text to the title bar of the dialog box.

Click the **Text Search** push button to search the list box for the text entered in the edit control and display the string that's found. To do this, the text must be read in a buffer in order to use it as lParam parameter for the CB\_SELECTSTRING message. The -1 value in the wParam parameter states that the search should start at the beginning. If the procedure doesn't find a string with the desired letters, it returns a value of CB\_ERR. A message box informs the user of the unsuccessful search. Otherwise, the window procedure of the combo box automatically displays the first string it finds in its edit control.

The three radio buttons are processed identically. The ID value of the clicked button is saved in the widIcon global variable so that the normal window function can prompt for it and set the appropriate icon.

The current setting of both check boxes isn't determined until the dialog box is ended with **OK**. Then the BM\_GETCHECK message is sent to both check boxes to prompt for the status. The result is saved in the bLERI and bRILE global variables, which are analyzed in the WM\_PAINT message.

Call the EndDialog function to delete the dialog box from the screen both for IDOK and IDCANCEL. The only difference is in the second parameter, which represents the return value of the dialog box call.

The .RC file contains a small menu consisting of a menu item and a dialog box that can be addressed using the #include... statement. We have already discussed the DIALOGBX.DLG file.

Since it is a callback function, the dialog procedure must be entered in the definition file by the EXPORTS statement.

## Message boxes

A message box looks like a simple dialog box. You don't have to define a message box in the .RC file and you don't need a dialog procedure. Instead, you can create it simply by calling MessageBox. A message box has a title bar and a static text. When necessary, Windows displays this text in multiple lines. If you don't specify a title, Windows displays the default title, "Error", in the title bar.

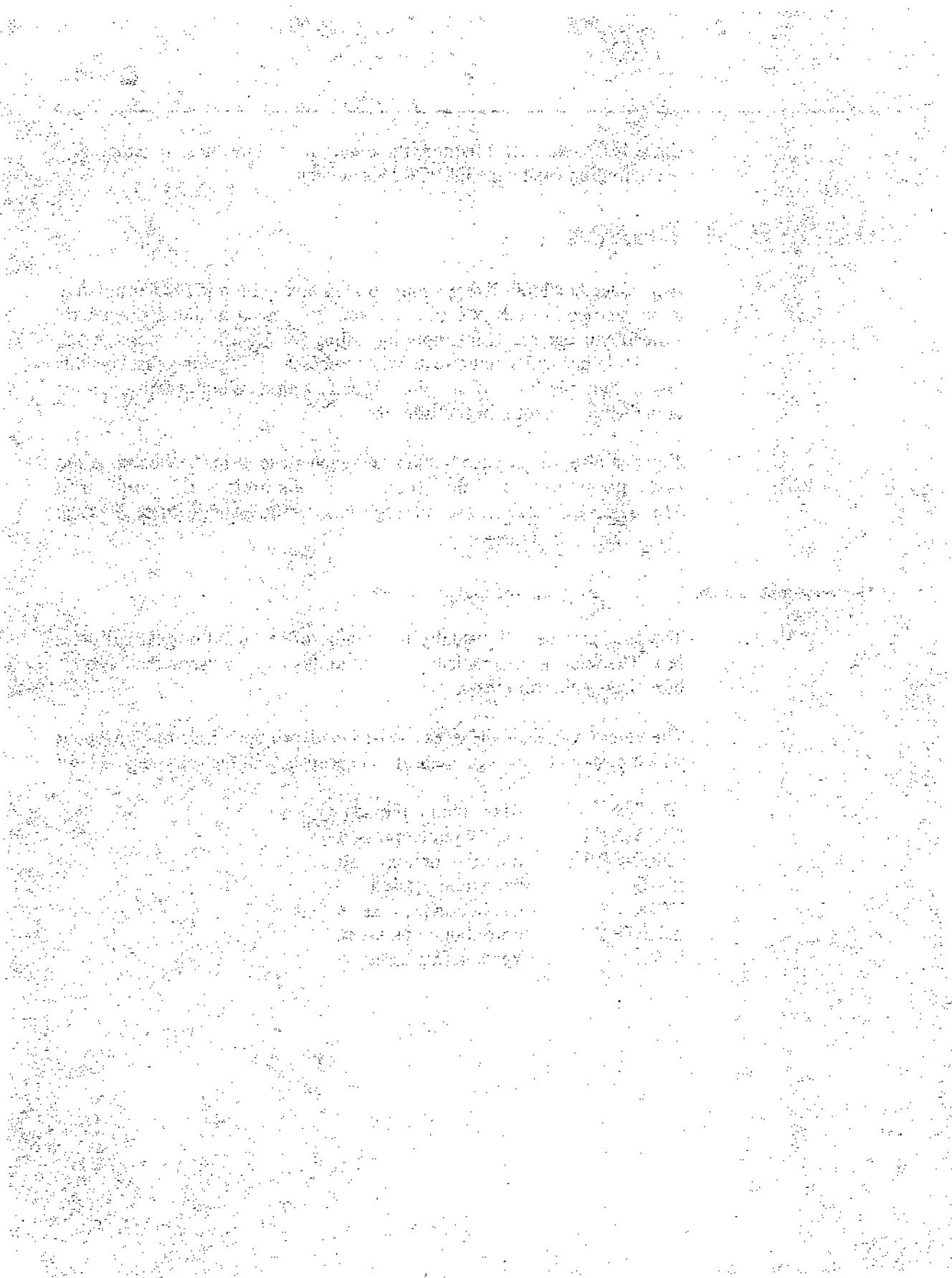
You can also assign push buttons and even an icon to the box. Use the last parameter to do this. For example, if you used MB\_YESNOCANCEL, the message box would contain three buttons: **Yes**, **No**, and **Cancel**.

```
int MessageBox( hWndParent, lpText, lpCaption, wType )
```

The first parameter is usually the handle of the window that calls the box. The focus is automatically passed to this window when the message box disappears from the screen.

The return value of the MessageBox call provides information about which push button was pressed. It can accept one of the following values:

IDABORT	Abort button pressed
IDCANCEL	Cancel button pressed
IDIGNORE	Ignore button pressed
IDNO	No button pressed
IDOK	OK button pressed
IDRETRY	Retry button pressed
IDYES	Yes button pressed



# Mapping Mode

## Overview

The GDI (Graphic Device Interface) of Windows first produces output in a logical environment before sending that output to the physical device.

The mapping mode defines the relationship between the character units in the logical environment and the pixels of a device. Therefore, the mapping mode that's set determines how the coordinate system used by the programmer is reproduced on the output device.

Four other parameters, which are closely linked to this mapping mode, are attributes of the device context just like mapping mode:

Parameter	Functions to change
Window Origin	SetWindowOrg
Window Extents	SetWindowExt
Viewport Origin	SetViewportOrg
Viewport Extents	SetViewportExt

The window parameters Window Origin and Window Extents set the origin of the logical coordinate system along with its extents. The two viewport parameters Viewport Origin and Viewport Extents are responsible for the device coordinates.

Mapping mode also determines the direction in which the values of the x and y axis increase. The origin (0,0) is usually the upper-left corner of the client area. The x values increase to the right while the y values increase toward the bottom.

Almost all of the functions that receive the handle of the device context as their first parameter work with logical coordinates. Therefore, they are dependent on the current mapping mode. For example, GetTextMetrics retrieves the letter dimensions of the current font based on the set mapping mode. In contrast to this, messages such as WM\_SIZE and WM\_MOUSEMOVE always work with device coordinates that have pixels for units. Windows uses two formulas to convert logical coordinates into device coordinates:

$$VpX = (WinX - WinOrgX) * \frac{VpExtX}{WinExtX} + VpOrgX$$

$$VpY = (WinY - WinOrgY) * \frac{VpExtY}{WinExtY} + VpOrgY$$

WinX and WinY are the x and y coordinates of the logical point that will be translated into the VpX and VpY device coordinates. The other eight parameters (WinOrgX, WinOrgY, WinExtX, WinExtY, VpOrgX, VpOrgY, VpExtX, and VpExtY) represent the coordinate origin and unit of the window or viewport.

These formulas can also be used to calculate in the other direction (device coordinates -> logical coordinates).

## Mapping mode types

There are eight different mapping modes:

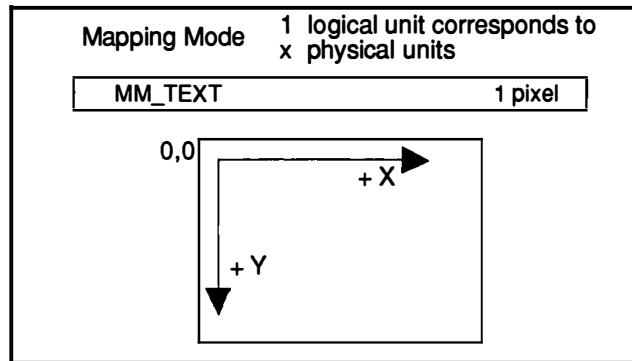
MM\_TEXT  
MM\_TWIPS  
MM\_LOMETRIC  
MM\_HIMETRIC  
MM\_LOENGLISH  
MM\_HIENGLISH  
MM\_ISOTROPIC  
MM\_ANISOTROPIC

Use SetMapMode to set each mapping mode. A set mapping mode remains current until another mode is selected or the device context is released. You can divide the modes into three groups.

## Device-dependent mode

This mode is called MM\_TEXT and is included in the device context as the default mode. It's generally used for simple text output because the y values increase from top to bottom, which follows the way we write (top to bottom).

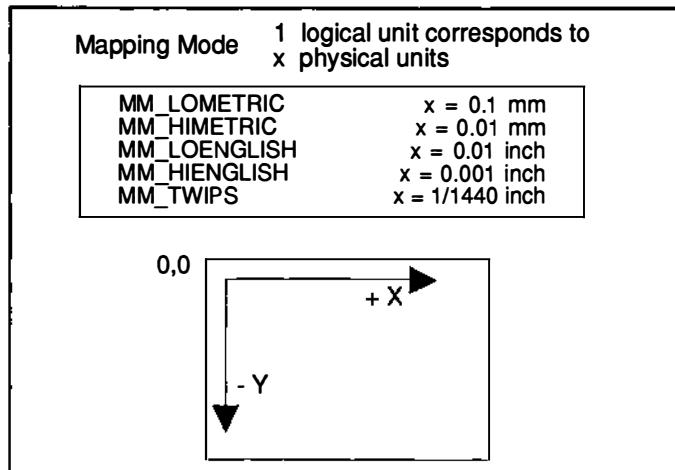
Since the unit is a pixel, whether a circle created with the `Ellipse(hDC, 20, 20, 80, 80)` call is displayed as a circle or as an ellipse depends on the device. If the pixel is square, a square will be displayed.



In this mode, only the coordinates of the window and the viewport can be changed, instead of the coordinate units, which are set to (1,1). For example, you could shift the origin of the coordinate system to the middle of the client area.

For reasons of clarity, you should change only one of the two origins.

## Metric modes



## *Mapping Mode*

---

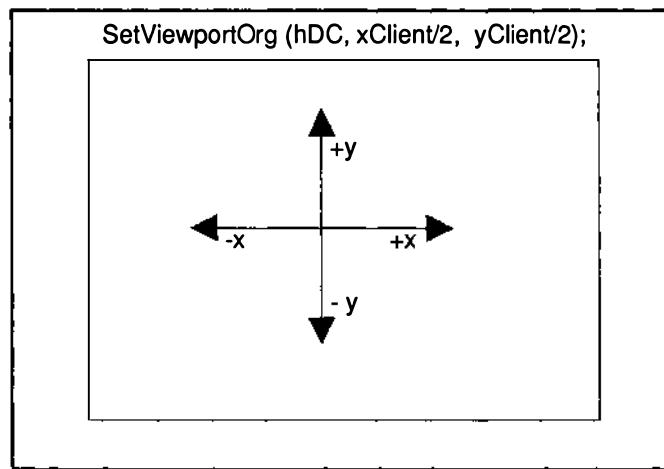
"Twip" is an abbreviation for "twentieth of a point". It is a unit of measurement used for printers. You can also work with millimeters or inches.

For example, if you set the MM\_LOMETRIC mapping mode and use the Ellipse(hDC, 20, 20, 80, 80) call to draw the circle, the result will always be a circle whose sides are  $60 * 0.1$  mm long. This is true regardless of your screen resolution.

The metric modes also differ from MM\_TEXT mode in the definition of the coordinate system. The origin is still in the upper-left corner, but the values of the y axis decrease towards the bottom.

It's easy to make a normal Cartesian coordinate system with four quadrants. Simply change the coordinate origin of the viewport or window. However, changing the origin of the window is a little more complicated because the coordinates of the client area must be converted into logical coordinates by using the DPtoLP function.

For example, the WM\_SIZE message determined the two values, xClient and yClient, which display the size of the client area.



The coordinate units are preset to fixed sizes in these modes and cannot be changed.

## Custom modes

The mapping modes MM\_ISOTROPIC and MM\_ANISOTROPIC are called custom modes because, along with the origin, you can also change the coordinate units of the window and the viewport. This allows you to create your own coordinate systems with any dimensions you like and then project the coordinate system into the client area.

In MM\_ISOTROPIC mode the graphic is reproduced exactly like the original (i.e., the same x:y ratio exists both in the logical and the physical system). As a result, part of the viewport might not be used in the output. The term isotropic means equal in all directions.

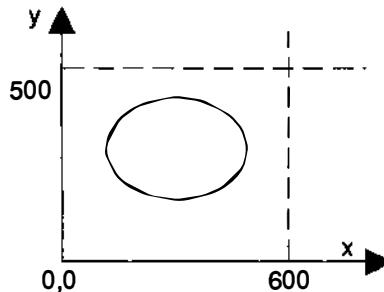
On the other hand, MM\_ANISOTROPIC mode attempts to make the best possible use of the viewport, even if this results in a distortion of the original. Frequently the coordinates of the viewport apply to the entire client area, whose current size can be determined with the GetClientRect function.

```
GetClientRect( hDC, &rect );
SetViewportOrg( hDC, rect.left, rect.top );
SetViewportExt( hDC, rect.right, rect.bottom );
```

When you set the coordinate units of the window, you must consider the direction of the axes in the custom coordinate system. The origin can be any of the four corners. The width of the window is the horizontal unit and the height of the window is the vertical unit.

## Mapping Mode

```
SetWindowOrg (hDC, 0,0);           SetWindowOrg (hDC, 0,500);
SetWindowExt (hDC, 600, 500);       SetWindowExt (hDC, 600, -500);
or
SetWindowOrg (hDC, 600,0);         SetWindowOrg (hDC, 600,500);
SetWindowExt (hDC, -600, 500);     SetWindowExt (hDC, -600, -500);
or
```



By defining the coordinate origin and unit of the window and viewport, you can retain the entire drawing in the client area when you reduce it.

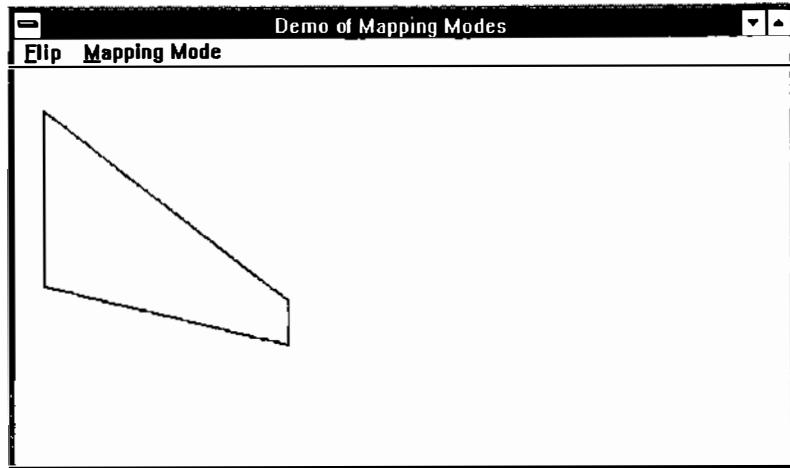
For example, you could use MM\_ANISOTROPIC mode to output text based on lines and columns instead of letter size. To do this, change the coordinate units of the window and the viewport.

```
SetMapMode( hDC, MM_ANISOTROPIC );
SetWindowExt ( hDC, 1, 1 );
SetViewportExt( hDC, xChar, yChar );

TextOut( hDC, 4, 5, "new text", 10 );
```

The two variables represent the height and average width of a letter. With the help of the TextOut function, the text then appears in the fifth line beginning at column four.

## Example of custom modes



This application's task is to reflect a simple graphic at parallels of the X and Y axis. The application then uses the two custom mapping modes with different values for the coordinate origin and units of the window. In addition, the application can display the graphic in a device-dependent mode and in a metric mode.

### New functions

`SetMapMode`  
`SetViewportExt`  
`SetViewportOrg`  
`SetWindowExt`  
`SetWindowOrg`

### Brief description

Sets the mapping mode  
 Sets the viewport extents  
 Sets the origin of the viewport  
 Sets the extents of the window  
 Sets the origin of the window

## Source code: MAPPING.C

```
/** MAPPING.C ****
** Demonstrates isotropic and anisotropic mapping modes
****/
```

```
#include "windows.h"                                // Include windows.h header file
#include "mapping.h"                                // Include mapping.h header file

int PASCAL WinMain(HANDLE, HANDLE, LPSTR, int);
long FAR PASCAL MappingWndProc(HANDLE, WORD, WORD, LONG);
```

## Mapping Mode

---

```
/** WinMain *****/
int PASCAL WinMain(hInstance, hPrevInstance, lpCmdLine, CmdShow)
    HANDLE hInstance;                                // Current instance
    HANDLE hPrevInstance;                            // Previous instance
    LPSTR lpCmdLine;                               // Long ptr to string after
                                                   // program name during execution
    int CmdShow;                                   // Specifies the application
                                                   // window's appearance
{

WNDCLASS wcMappingClass;                         // Window class declaration
HWND hWnd;                                       // Window handle
MSG msg;                                         // Message variable

if(!hPrevInstance)                                // Initialize first instance
{

/** Specify window class *****/
    wcMappingClass.style      = CS_HREDRAW | CS_VREDRAW;
    wcMappingClass.lpfnWndProc = MappingWndProc;
    wcMappingClass.cbClsExtra = 0;
    wcMappingClass.cbWndExtra = 0;
    wcMappingClass.hInstance = hInstance;
    wcMappingClass.hIcon     = LoadIcon(NULL, IDI_APPLICATION);
    wcMappingClass.hCursor   = LoadCursor(NULL, IDC_ARROW);
    wcMappingClass.hbrBackground = GetStockObject(WHITE_BRUSH);
    wcMappingClass.lpszMenuName = "Mapping";
    wcMappingClass.lpszClassName = "Mapping";

    if(!RegisterClass(&wcMappingClass))
        return FALSE;
}

/** Specify appearance of application's main window *****/
hWnd = CreateWindow("Mapping",
                    "Demo of Mapping Modes",
                    WS_OVERLAPPEDWINDOW,
                    CW_USEDEFAULT, 0,
                    CW_USEDEFAULT,0,
                    NULL,
                    NULL,
                    hInstance,
                    NULL);
```

```

ShowWindow(hWnd, CmdShow);           // Make window visible
UpdateWindow(hWnd);                // Update window

while( GetMessage(&msg, NULL, 0, 0) ) { // Message reading
    TranslateMessage(&msg);        // Message translation
    DispatchMessage (&msg);       // Send message to Windows
}
return msg.wParam;                  // Return wParam of last message
}

/** MappingWndProc *****/
/** Main window function: All messages are sent to this window      */
/* *****/
long FAR PASCAL MappingWndProc(hWnd, message, wParam, lParam)
{
    HANDLE          hWnd;
    WORD            message;
    WORD            wParam;
    LONG            lParam;

    HDC             hDC;
    PAINTSTRUCT     ps;

    static HMENU     hMenu;
    static WORD      wFigurOld, wMMOld;
    static int       iMapping;
    RECT            rect;

    static int       sWinOrgX,sWinOrgY,
                    sWinExtX,sWinExtY;

    struct
    {
        int      WinOrgX;
        int      WinOrgY;
        int      WinExtX;
        int      WinExtY;
    }    WinPoints [4] = { 0,200, 200,-200,
                        200,200, -200,-200,
                        0, 0, 200, 200,
                        200, 0, -200, 200};

    int      map [4] = {MM_TEXT, MM_ISOTROPIC,
                      MM_ANISOTROPIC, MM_LOMETRIC};
}

```

## Mapping Mode

---

```
POINT FourSider [5] = {{ 20, 150},
                        { 20,  30},
                        {190, 160},
                        {190, 190},
                        { 20, 150}};

switch (message)
{
    case WM_CREATE:           // Draw window
        hMenu = GetMenu(hWnd);
        iMapping = MM_TEXT;
        wMMOld = ID_DEF;
        wFigurOld = ID_ORIGIN;

        sWinOrgX = 0;
        sWinOrgY = 200;
        sWinExtX = 200;
        sWinExtY = -200;
        break;

    case WM_COMMAND:
        switch(wParam)      // Respond to menu items
        {
            case ID_ORIGIN:
            case ID_RIGHT:
            case ID_DOWN:
            case ID_LEFT:
                sWinOrgX = WinPoints [wParam -
                                      ID_ORIGIN].WinOrgX;
                sWinOrgY = WinPoints [wParam -
                                      ID_ORIGIN].WinOrgY;
                sWinExtX = WinPoints [wParam -
                                      ID_ORIGIN].WinExtX;
                sWinExtY = WinPoints [wParam -
                                      ID_ORIGIN].WinExtY;

                CheckMenuItem(hMenu, wFigurOld,
                               MF_UNCHECKED);
                CheckMenuItem(hMenu, wParam, MF_CHECKED);
                wFigurOld = wParam;
                InvalidateRect(hWnd, NULL, TRUE);
                break;

            case ID_DEF: // Default mode
            case ID_ISO: // Isotropic mode
            case ID_ANI: // Anisotropic mode
            case ID_LOM: // Lometric mode
```

```
iMapping = map [wParam - ID_DEF];
CheckMenuItem(hMenu, wMMOld, MF_UNCHECKED);
CheckMenuItem(hMenu, wParam, MF_CHECKED);
InvalidateRect (hWnd, NULL, TRUE);
wMMOld = wParam;
break;

default:
    break;
}

case WM_PAINT:           // Client area redraw
hDC = BeginPaint(hWnd, &ps);

SetMapMode (hDC, iMapping);

GetClientRect (hWnd, &rect);

if ((iMapping == MM_ISOTROPIC)
    || (iMapping == MM_ANISOTROPIC))
{
    SetWindowOrg (hDC, sWinOrgX, sWinOrgY);
    SetWindowExt (hDC, sWinExtX, sWinExtY);

    SetViewportOrg (hDC, rect.left, rect.top);
    SetViewportExt (hDC, rect.right, rect.bottom);
}

if (iMapping == MM_LOMETRIC)
    SetViewportOrg (hDC, 0, rect.bottom);

Polygon (hDC, FourSider, 4);

EndPaint (hWnd, &ps);
break;

default:
    return (DefWindowProc (hWnd, message, wParam, lParam));
break;
}
return (0L);
}
```

## Module definition file: MAPPING.DEF

```
NAME      Mapmode
DESCRIPTION 'Example of mapping modes'
EXETYPE   WINDOWS
STUB      'WINSTUB.EXE'
CODE      PRELOAD MOVEABLE
DATA      PRELOAD MOVEABLE MULTIPLE
HEAPSIZE  1024
STACKSIZE 4096
EXPORTS   MappingWndProc    @1
```

## Resource script: MAPPING.RC

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

Mapping MENU
BEGIN
    POPUP "&Flip"
    BEGIN
        MENUITEM "&Original",           ID_ORIGIN, CHECKED
        MENUITEM "&Right",             ID_RIGHT
        MENUITEM "&Down",              ID_DOWN
        MENUITEM "Down &and Right",   ID_LEFT
    END

    POPUP "&Mapping Mode"
    BEGIN
        MENUITEM "&Default",          ID_DEF,      CHECKED
        MENUITEM "&Isotropic",        ID_ISO
        MENUITEM "&Anisotropic",       ID_ANI
        MENUITEM "&Lometric",         ID_LOM
    END
END
```

## Header file: MAPPING.H

```
#define ID_ORIGIN      20
#define ID_RIGHT       21
#define ID_DOWN        22
#define ID_LEFT        23

#define ID_DEF         10
#define ID_ISO          11
#define ID_ANI          12
#define ID_LOM          13
```

To compile and link this application, use the RCOMPILE.BAT batch file described throughout this chapter. Here's the listing again:

```
cl -c -Gw -Zp %1
rc -r %1.rc
link /align:16 %1,%1.exe,,libw+slibcew,%1.def
rc %1.res
```

Save this file to a directory contained in your path or one containing the source, module definition and resource scripts. Type the following and press **Enter** to compile MAPPING:

```
rcompile mapping
```

## How MAPPING.EXE works

You can set the different mapping modes from a pop-up menu. There is a different pop-up menu for the flipping, which can only be used with the custom modes (Anisotropic and Isotropic).

When the WM\_CREATE message is being processed, initializations are executed by setting different variables. At the beginning of run time, the MM\_TEXT mapping mode is the current mqde and the square is displayed without a reflection.

Because of the application's structure, all the function references linked with the mapping mode appear in the WM\_PAINT message as it's being processed.

This happens because the necessary handle to the device context is also present. In the other messages, only those variables are filled.

During processing, the four menu items from the **Flip** pop-up menu can be combined. The correct values for the origin and the units of the window can be taken from the array, which has a user-defined structure.

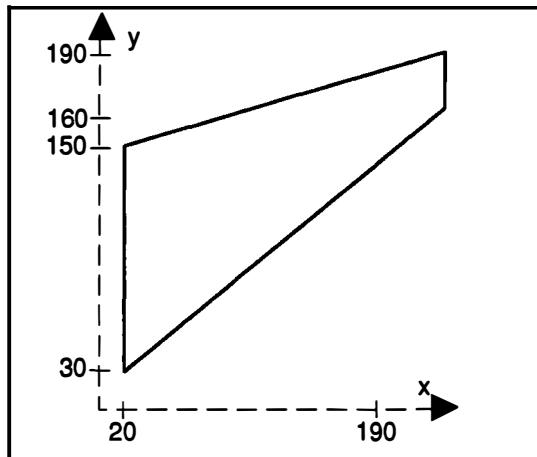
The ID values of the four menu items were defined in ascending order in the MAPPING.H header file. The index to the array is then the difference between the current menu ID, which is in wParam, and the ID\_ORIGIN menu ID.

Then the checkmark must be set at the new item and deleted from the old item. To display the new setting, the InvalidateRect function creates a WM\_PAINT message.

If you select one of the menu items from the **Mapping Mode** pop-up menu, the processing is quite similar to the menu items in the **Flip** menu. The appropriate mapping mode is taken from the map array, whose index is again determined from the menu ID. The application must change the checkmark and create the WM\_PAINT message.

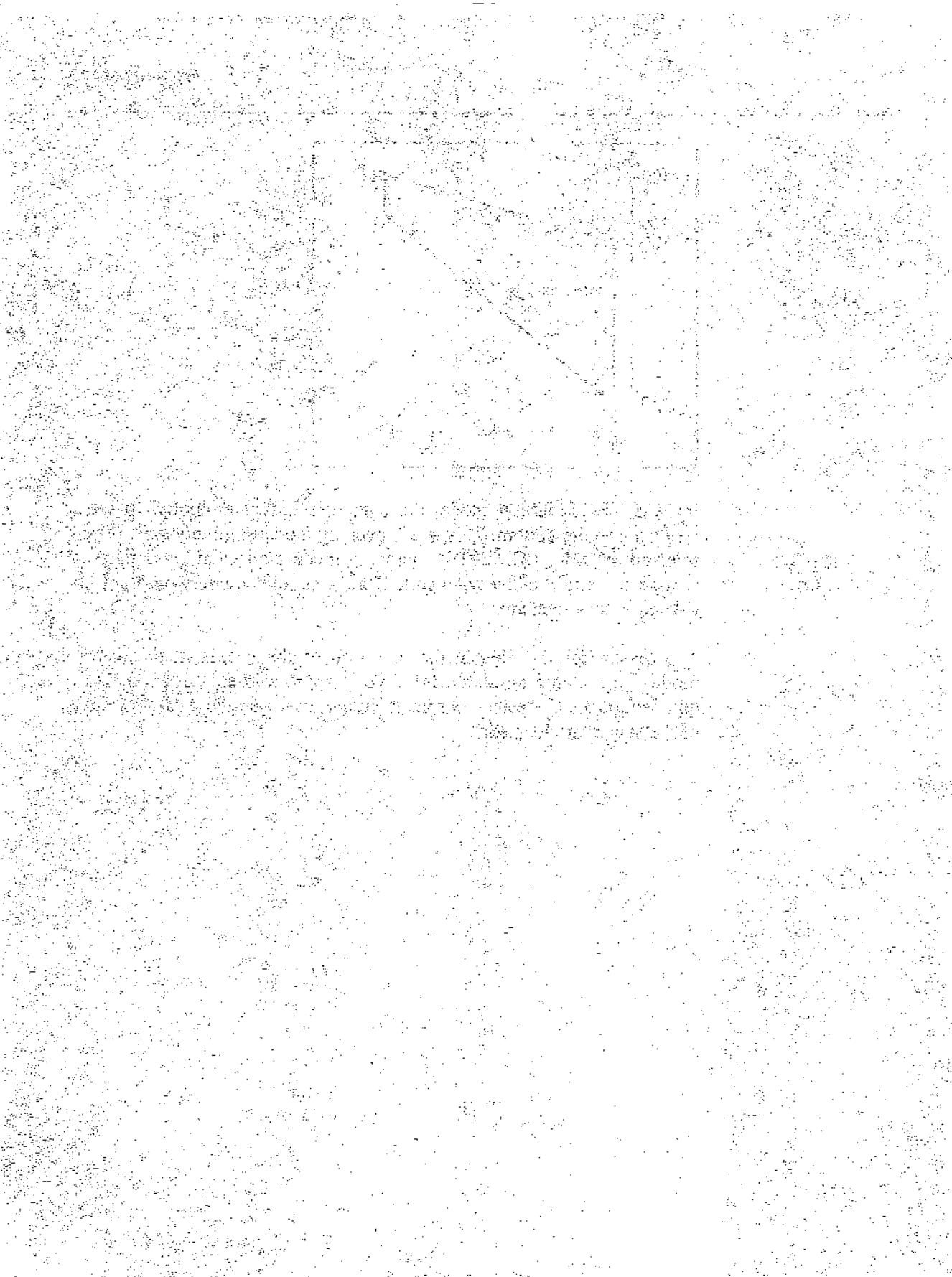
In WM\_PAINT, the mapping mode and the coordinate origin and units for the MM\_ISOTROPIC and MM\_ANISOTROPIC modes can now be set for both the window and the viewport. The GetClientRect function determines the values for the viewport.

The logical coordinate system looks as follows:



In MM\_TEXT default mode, the square is drawn as though it were reflecting to the bottom. This is achieved by increasing the y axis. If you selected the MM\_LOMETRIC mapping mode, you would also have to change the origin of the viewport. Otherwise, the square would not be located in the client area.

The reason for this lies in the direction of the y axis in the metric modes. By changing the origin, you are now displaying the first quadrant of the Cartesian coordinate system in which the positive values of the square are located.



# Bitmaps

## Overview

Bitmaps are graphics that are used in various ways. With bitmaps, you can draw graphics that can be used for all kinds of applications.

If you're using pattern brushes, you need bitmaps (refer to the chapter on Output, which appeared earlier in this book).

Windows Version 3.0 and higher recognize two types of bitmaps:

device-dependent bitmaps  
device-independent bitmaps (DIB)

A device-dependent bitmap consists of a bit pattern that's placed in memory and sent to an output device by specific functions. These bitmaps have a special relationship between the bits in memory and the pixels that appear on the device.

However, a device-independent bitmap (DIB), doesn't have a device specific format. DIB functions translate the device-independent graphic data into a format that the current output device can understand.

## Device-dependent bitmaps

A device-dependent bitmap is a direct copy of a section of the screen memory. Its structure varies with the graphics card being used. This is where the term "device-dependent" originates.

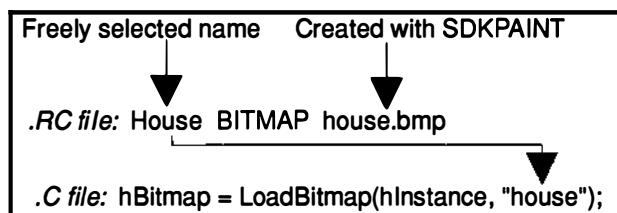
## Creating a bitmap

Device-dependent bitmaps can be created by using one of the following methods:

Bitmap as resource  
Empty bitmap  
Bitmap with bit array

## Creating a bitmap as a resource:

Bitmaps can be drawn by using a Windows-based drawing application, such as SDKPaint or Paintbrush. SDKPaint generates bitmaps up to 72 pixels by 72 pixels. The finished graphic must be saved as a bitmap file (i.e., with a .BMP extension). These filenames are passed to the resource script file by the BITMAP resource script statement (refer to the chapter on resources for more information).



The LoadBitmap function needs the name so that it can load the bitmap into memory and return the handle needed for output.

Bitmaps can also be created without resources. You can do this either with GDI functions or a bit array. One of four different functions can be used to create an empty bitmap that can then be filled using one of the methods discussed earlier.

```
HBITMAP CreateBitmap( nWidth, nHeight, nPlanes, nBitCount, lpBits )
HBITMAP CreateBitmapIndirect( lpBitmap )
HBITMAP CreateCompatibleBitmap( hDC, nWidth, nHeight )
HBITMAP CreateDiscardableBitmap( hDC, nWidth, nHeight )
```

The parameters are different for the first two functions. The HBITMAP CreateBitmap function passes parameters directly, while the HBITMAP CreateBitmapIndirect function uses a pointer that indicates individual values to the BITMAP structure.

First let's discuss HBITMAP CreateBitmap. Color bitmaps can be divided into two types. The first type uses color planes, in which each color plane represents a color. The color of the specified pixels in each plane is defined as a bit. The planes are then placed behind one another in memory.

If an output device supports eight colors, three color planes implement the colors (one red plane, one green plane, and one blue plane). A

yellow pixel actually contains one bit from the green plane and one bit from the blue plane. To use this method, a value of 3 must be in the nPlanes (number of planes) parameter, and a value of 1 must be in the nBits (number of bits per pixel) parameter.

The second method uses only one color plane. Multiple bits within this single color plane control the colors. For example, an eight color display requires three bits (i.e., three bits define each pixel). In this method, the nPlanes parameter requires a value of 1 and the nBits parameter requires a value of 3. The method used depends on the output device you select.

Let's look at the layout of a 4 pixel by 4 pixel, 8 color bitmap:

## Bitmap appearance

Line 1: Magenta

Line 2: Yellow

Line 3: Blue

Line 4: Red

## Bitmap coding

1 bit per pixel	3 bits per pixel	
1 color plane	or	3 color planes

1 1 1 1	101 101 101 101
1 1 1 1	110 110 110 110
0 0 0 0	001 001 001 001
1 1 1 1	100 100 100 100
0 0 0 0	
1 1 1 1	
0 0 0 0	
0 0 0 0	

1 1 1 1
0 0 0 0
1 1 1 1
0 0 0 0

The `CreateCompatibleBitmap` and `CreateDiscardableBitmap` functions don't include the `nPlanes` and `nBitCount` parameters. Windows passes the options specified in the device context as a handle to the first parameter.

A bitmap created by the `CreateDiscardableBitmap` function can be removed from memory if a device context isn't selected. `SelectObject` function attempts to select this bitmap at a later time; this function returns a value of 0.

## **Creating bitmaps with GDI functions**

The empty bitmap can then be filled using GDI output functions, such as `rectangle`. This bitmap must be placed in a memory device context (see the example accompanying the description of the `PatBlt` function).

## **Creating bitmaps with a bit array**

If you use the `CreateBitmap` function to create a new bitmap, the last parameter can pass a pointer to an array containing a predefined bit pattern. This array can then be used to initialize the bitmap. Otherwise, a bitmap can be filled using the `SetBitmapBits` function.

## **Output functions**

Three functions can be used to output device-dependent bitmaps:

`PatBlt`  
`BitBlt`  
`StretchBlt`

The term "Blt" (pronounced "blit") is an abbreviation for block transfer. The `GetDeviceCaps` function and `RASTERCAPS` parameter determine whether the output device supports these three bitmap functions.

```
if( GetDeviceCaps( hDC, RASTERCAPS ) & RC_BITBLT )
    TRUE => Device can display bitmaps
```

If the return value matches `RC_BITBLT`, this device supports device-dependent bitmaps. The second and third functions require an additional device context (usually a memory device context). The bitmap is written

to this device context, with the upper-left corner of the bitmap corresponding to the upper-left corner of the memory device context.

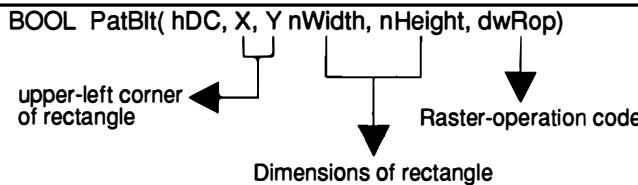
Finally, the bitmap can be copied from its source (the memory device context) to a target device context (e.g., display context), then displayed on the output device. When the output ends, the memory device context releases the bitmap and returns to its original status.

```
HDC hDC, hMemDC;
HBITMAP hNewBitmap, hOldBitmap;
hDC = GetDC( hWnd);
hNewBitmap = ....;
hMemDC = CreateCompatibleDC( hDC);
hOldBitmap = SelectObject( hMemDC, hNewBitmap);

Bitmap-Output function

SelectObject( hMemDC, hOldBitmap);
DeleteDC( hMemDC);
ReleaseDC( hWnd, hDC);
```

This sequence of functions isn't always used. For example, a bitmap can be created without the device context. The PatBlt function sends a bit pattern to the device using an existing handle for a device context. However, other functions need only a device context.



The second through fourth parameters indicate which range should be written to the output device. The parameters are in logical units. The pattern drawn in this rectangle is a combination of the current brush (stored in the device context) and the pattern (already prepared for the output device). The last parameter (the raster operation code) dictates how these bits interact.

For example, if you pass the WHITENESS parameter, the entire rectangle is filled with white. In this case, neither the current brush nor the available pattern are needed. This parameter generates an empty bitmap when used with one of the four functions previously described.

## *Bitmaps*

---

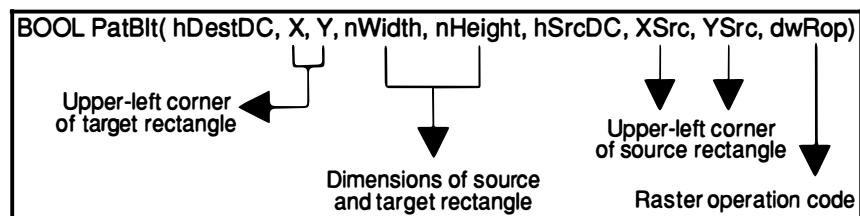
This empty bitmap can be used to delete the contents of an undefined new bitmap.

```
HDC hDC, hMemDC;  
HBITMAP hNewBitmap, hOldBitmap;  
  
hNewBitap = CreateBitmap( 16, 16, 1, 1, NULL );  
  
hDC = GetDC( hWnd );  
hMemDC = CreateCompatibleDC( hDC );  
hOldBitmap = SelectObject( hMemDC, hNewBitmap );  
  
PatBlt( hMemDC, 0, 0, 16, 16, WHITENESS );  
  
SelectObject( hMemDC, hOldBitmap );  
DeleteDC( hMemDC );  
Release( hWnd, hDC );
```

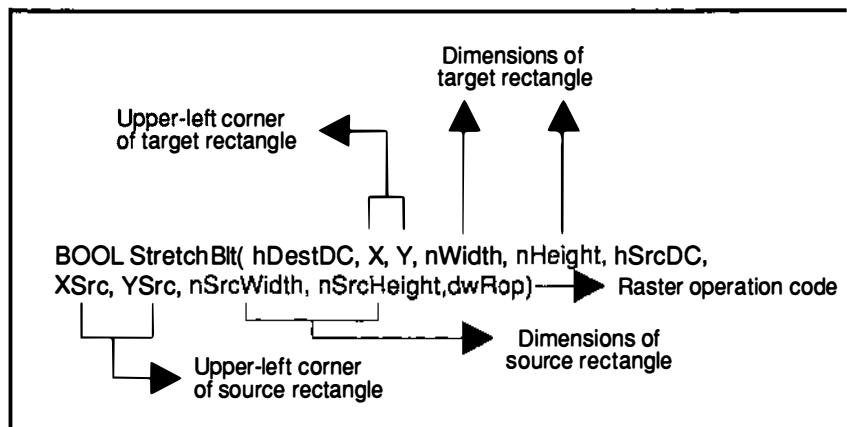
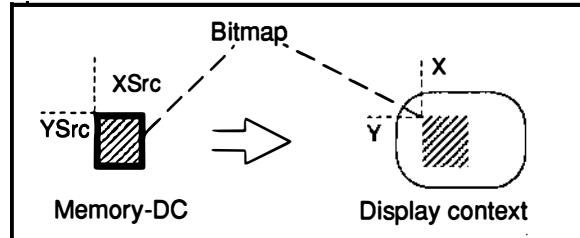
This function sequence is very similar to the pattern that applies to the BitBlt and StretchBlt functions (see above). The same results can be achieved using the PATCOPY raster operation code which fills the specified rectangle with the current brush.

```
hNewBrush = GetStockObject( WHITE_BRUSH );  
hOldBrush = SelectObject( hMemDC, hNewBrush );  
  
PatBlt( hMemDC, 0, 0, 16, 16, PATCOPY );  
  
SelectObject( hMemDC, hOldBrush );
```

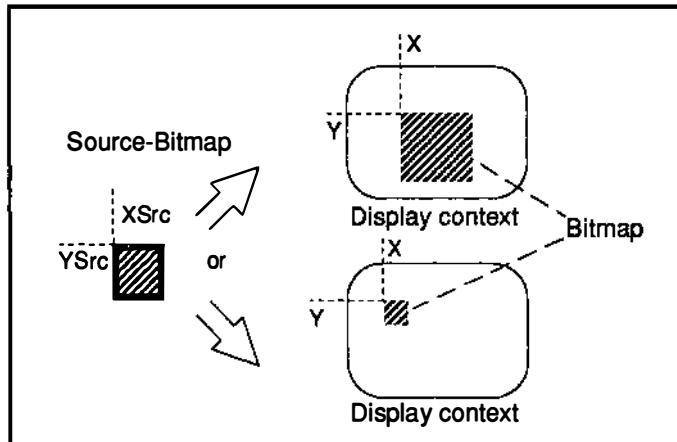
The BitBlt function transfers a bitmap from the source device context (identified by the hSrcDC handle and a memory device context) to a target device context. Since the size of the bitmap doesn't change, it is applied only once.



The x and y coordinates of the source and target rectangles correspond to the upper-left corner of the rectangle. The last parameter gives the raster operation code to PatBlt and other functions.



The StretchBlt function (see the above illustration) is needed if the bitmap must be resized in the target device context. StretchBlt consists of two parameters indicating the new height and width of the target rectangle.



This function can also mirror the bitmap, expressing the width and height values as negative numbers.

Reducing a bitmap means combining several lines or columns into one line or column. This eliminates some information about the bitmap's appearance. However, stretching mode adds pixel combinations. The SetStretchBltMode function activates the desired mode. Three options can be used:

BLACKONWHITE  
WHITEONBLACK  
COLORONCOLOR

BLACKONWHITE mode is the default, in which logical AND operations add groups of pixels as needed. This means that the resulting pixels are white only if all the pixels of the original are white. If you select WHITEONBLACK mode, white dominates through a logical OR operation.

Black pixels exist only if all the pixels of the original are black. If you reduce a color bitmap, you should use COLORONCOLOR mode. In this case, instead of removing combinations through the StretchBlt function, only select lines or columns are removed. Otherwise, some undesirable color effects could occur.

The raster operation code used by all three bitmap output functions defines how GDI combines the target device context's current brush

---

colors with the pixels in both the source and target rectangles. The result is stored in the target rectangle.

The logical combinations of three sizes provides 256 possible combinations. The fifty most practical combinations are stored under one name. This name is defined in the WINDOWS.H include file, and contains values such as SRCINVERT, PATCOPY, BLACKNESS, and SRCCOPY.

The most frequently used value is SRCCOPY. This raster operation code applies only to the appearance of the source bitmap.

## More about bitmaps

The SetPixel function sets a pixel, in a specific color, in a specific location on the screen. However, Windows developers rarely use this function. Instead, they use the GetPixel function to control pixel colors.

The GetObject function provides the size of a bitmap to fill the logical bitmap structure with type BITMAP.

bmType	For logical bitmaps: 0
bmWidth	Bitmap width in pixels
bmHeight	Bitmap height in raster lines
bmWidthBytes	Number of bytes in a raster line
bmPlanes	Number of color planes
bmBitsPixel	Number of color bits per pixel
bmBits	Pointer to a byte array

The above values specify bitmap height and width, as well as the number of color planes and bits per pixel. However, the function doesn't fill in the last array. The GetBitmapBits function must be called to find the bit pattern itself.

```
BITMAP bm;
BYTE Bits[200];
HBITMAP hBitmap;

hBitmap = LoadBitmap( hInstance, "Koala" );
GetObject( hBitmap, sizeof( BITMAP ), (LPSTR) &bm );
GetBitmapBits( hBitmap, bm.bmHeight * bm.bmWidthBytes, Bits );
```

This example generates a bitmap with only one color plane. If you want to add color planes, the second parameter must be multiplied by the number of planes.

A monochrome bitmap can be displayed in color if you specify the foreground and background colors. The foreground color represents the set (white) bits, while the background color represents the unset (black) bits.

```
SetTextColor( hDC, RGB( 0, 0, 255 ) );
SetBkColor( hDC, RGB( 255, 0, 0 ) );
```

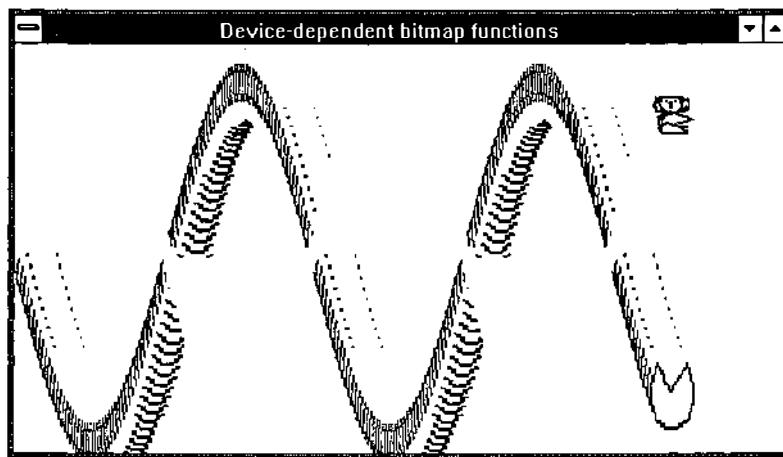
The following illustration shows a sample monochrome bitmap:

11111111	1 => blue
10000001	0 => red
10000001	
10000001	
11111111	

If you create a monochrome bitmap with BitBlt or StretchBlt, based on the above information, the colors default to a blue border and red foreground.

When the bitmap has served its purpose (or the application ends), the bitmap should be deleted, and the memory occupied by the bitmap released. Also, the bitmap shouldn't be included in any device context. The DeleteObject function performs this task.

## Example of a device-dependent bitmap



This example uses two bitmaps, which move in mirrored sine formation over the client area. One bitmap looks like a koala, based on information loaded from the .RC file.

The second bitmap, generated by the Pie GDI function, increases in size. The WM\_TIMER message controls the continuous movement.

### New messages

**WM\_SIZE**

### New functions

BitBlt

CreateBitmap

CreateCompatibleDC

DeleteDC

GetObject

PatBlt

StretchBlt

### Brief description

Sends size changes to window

### Brief description

Copies bitmap from source device context to target device context

Creates a bitmap

Creates a memory device context

Deletes a memory device context

Copies logical object data in buffer

Creates a bit pattern

Copies bitmap from source device context to target device context for eventual resizing

### New structures

### Brief description

BITMAP

Bitmap data structure

## Source code: BIT1.C

```
/** BIT1.C ****
** Demonstrates device-dependent bitmap functions
****

#include "windows.h"           // Include windows.h header file
#include "math.h"               // Include math.h header file

BOOL Bit1Init ( HANDLE );
long FAR PASCAL Bit1WndProc( HWND, unsigned, WORD, LONG);

/** WinMain ****

int PASCAL WinMain( hInstance, hPrevInstance, lpszCmdLine, cmdShow )
HANDLE  hInstance, hPrevInstance;           // Current & previous instance
LPSTR   lpszCmdLine;                      // Ptr to string after prg name
int     cmdShow;                          // App. window's appearance
{
    MSG      msg;                         // Message handle
    HWND     hWnd;                        // Window handle

    if (!hPrevInstance)                  // Initialize first instance
    {
        if (!Bit1Init( hInstance ))
            return FALSE;
    }

    hWnd = CreateWindow("Bit1", "Device-dependent bitmap functions",
                       WS_OVERLAPPEDWINDOW,CW_USEDEFAULT, 0, CW_USEDEFAULT, 0,
                       NULL, NULL, hInstance, NULL);

    while (!SetTimer(hWnd, 1, 100, NULL))
        if (IDCANCEL == MessageBox(hWnd, "Too many timers running",
                                    "Timer", MB_RETRYCANCEL))
            return FALSE;

    ShowWindow( hWnd, cmdShow );
    UpdateWindow( hWnd );

    while (GetMessage(&msg, NULL, 0, 0)) // Message reading
    {
        TranslateMessage(&msg);          // Message translation
        DispatchMessage(&msg);          // Send message to Windows
    }

    return (int)msg.wParam;                // Return wParam of last message
}
```

```
}

/** Specify window class *****/
BOOL Bit1Init( hInstance )
HANDLE hInstance;
{
    WNDCLASS      wcBit1Class;

    wcBit1Class.hCursor      = LoadCursor( NULL, IDC_ARROW );
    wcBit1Class.hIcon        = LoadIcon( NULL, IDI_APPLICATION );
    wcBit1Class.lpszMenuName = NULL;
    wcBit1Class.lpszClassName = "Bit1";
    wcBit1Class.hbrBackground = GetStockObject( WHITE_BRUSH );
    wcBit1Class.hInstance     = hInstance;
    wcBit1Class.style         = CS_HREDRAW | CS_VREDRAW;
    wcBit1Class.lpfnWndProc   = Bit1WndProc;
    wcBit1Class.cbClsExtra    = 0 ;
    wcBit1Class.cbWndExtra    = 0 ;

    if (!RegisterClass( &wcBit1Class ) )
        return FALSE;

    return TRUE;
}

/** Bit1WndProc *****/
/** Main window function: All messages are sent to this window      */
/** *****/
long FAR PASCAL Bit1WndProc( hWnd, message, wParam, lParam )
HWND      hWnd;
unsigned message;
WORD      wParam;
LONG      lParam;
{
    HANDLE      hInst;
    PAINTSTRUCT ps;
    HDC         hDC, hMem;
    static HBITMAP hBit1, hBit2, hOld;
    static BITMAP bm;
    static BYTE   Bits[200];
    static int    xPos, y1Pos, y2Pos, i;
    static int    xClient, yClient;
    static double y, x;

    switch (message)
{
```

```
case WM_CREATE: // Create window
    hInst = GetWindowWord( hWnd, GWW_HINSTANCE);
    hBit1 = LoadBitmap(hInst, "koala");
    GetObject(hBit1, sizeof(BITMAP), (LPSTR)&bm);

    hBit2 = CreateBitmap(20,20, 1,1, NULL);
    hDC = GetDC(hWnd);
    hMem = CreateCompatibleDC(hDC);
    hOld = SelectObject(hMem, hBit2);
    PatBlt( hMem, 0,0, 20,20, WHITENESS);
    Pie(hMem, 0,0, 20,20, 2,0, 18,0);
    SelectObject(hMem, hOld);
    DeleteDC(hMem);
    ReleaseDC(hWnd, hDC);
    break;

case WM_SIZE:
    xClient = LOWORD(lParam);
    yClient = HIWORD(lParam);
    yClient = yClient - bm.bmHeight;
    y1Pos = yClient/2;
    y2Pos = yClient/2;
    x = xPos = 0;
    y = 0.0;
    i = 0;
    break;

case WM_TIMER: // Timer running?
    hDC = GetDC(hWnd);
    // PatBlt draws BMP in background color
    PatBlt(hDC, xPos, y1Pos, bm.bmWidth,
            bm.bmHeight, WHITENESS);

    x = i * 3.14159/50;
    xPos = i * (xClient/200);
    i++;
    if (xPos > xClient)
        i = 0;
    y = sin(x);

    hMem = CreateCompatibleDC(hDC);

    // Draw koala bitmap

    hOld = SelectObject(hMem, hBit1);
    y1Pos = (int)(-y * ((double)yClient)/2 +
                  ((double)yClient)/2);
```

```

        BitBlt(hDC, xPos, y1Pos, bm.bmWidth,
                bm.bmHeight, hMem, 0,0, SRCCOPY);

                // Enlarge pie bitmap

        SelectObject(hMem, hBit2);
        y2Pos = (int)(y * ((double)yClient)/2 +
                      ((double)yClient)/2);
        StretchBlt(hDC, xPos, y2Pos, 30,50,
                    hMem,0,0, 20,20, SRCCOPY);

        SelectObject(hMem, hOld);
        DeleteDC(hMem);
        ReleaseDC(hWnd, hdc);

        break;

    case WM_DESTROY:           // Destroy window
        KillTimer( hWnd, 1);
        DeleteObject(hBit1);
        DeleteObject(hBit2);
        PostQuitMessage(0);
        break;

    default:
        return (DefWindowProc( hWnd, message, wParam, lParam ));
        break;
    }
    return(0L);
}
}

```

## Module definition file: BIT1.DEF

```

NAME      Bit1

DESCRIPTION 'Device-dependent bitmap functions'

EXETYPE   WINDOWS

STUB      'WINSTUB.EXE'

CODE      PRELOAD MOVEABLE
DATA      PRELOAD MOVEABLE MULTIPLE

```

```
HEAPSIZE      4096  
STACKSIZE     4096  
  
EXPORTS       Bit1WndProc
```

## Resource script: BIT1.RC

```
koala          BITMAP        koala.bmp
```

To compile and link this application, use the RCOMPILE.BAT batch file described earlier in this book. Here's the listing again:

```
cl -c -Gw -Zp %1  
rc -r %1.rc  
link /align:16 %1,%1.exe,,libw+slibcew,%1.def  
rc %1.res
```

Save this file to a directory contained in your path or one containing the source, module definition and resource scripts. Type the following and press **e** to compile BIT1:

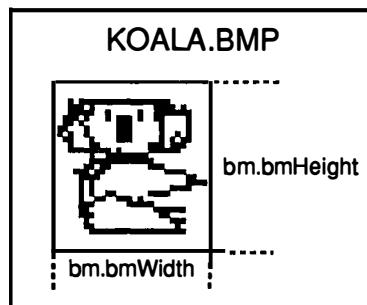
```
rcompile bit1
```

## How BIT1.EXE works

The SetTimer function in the WinMain routine starts the timers. If all 16 system timers are being used, additional timers cannot be activated. The application displays a dialog box indicating this.

The dialog box contains buttons that enable the user to retry or cancel the operation. Clicking **Retry** attempts to start the timer again.

The WM\_CREATE message provides the handles for both bitmaps. The KOALA bitmap was created by using SDKPAINT.EXE and is saved as KOALA.BMP. The resource script specifies this file as a resource.

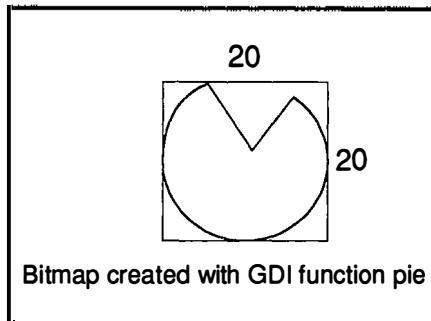


KOALA is the name of the resource as specified in the second parameter of the LoadBitmap function. The GetObject function transfers the bitmap dimensions and other values to the bm variable of the BITMAP structure.

The second bitmap is created by the CreateBitmap function. This bitmap is 20 pixels high by 20 pixels wide. A memory device context must be selected by the bitmap to permit access to this bitmap.

The CreateCompatibleDC function provides the handle needed for this memory device context. Later, the DeleteDC function releases this handle.

The bitmap contains random values. This prevents the CreateBitmap function from initializing as an array. Therefore, the entire graphic is filled with white (PatBlt function) before the Pie GDI function executes.



The WM\_SIZE message indicates when the window size changes. The lParam parameter contains the new width in the low word of the value, and the new height in the high word of the value.

The KOALA bitmap receives its height from the window height.

The result acts as a basis for computing the y value when both bitmaps are drawn. This output occurs through the WM\_TIMER message over the course of 0.1 second.

The old KOALA bitmap is deleted from the client area before it can be redrawn in the new position. The PatBlt function realizes this by drawing the old bitmap in the same color as the background.

The sine curve requires x values between 0 and 2p. Two sine curves should appear in the client area. The bitmaps are drawn a total of 200 times. When they reach the right border and are removed, the x position resets to a value of 0.

The output itself requires a memory device context from which the bitmaps are selected. This device context is sometimes the source device context. The target device context is represented by the display context of the client area.

The result of the sine calculation must be moved to the x axis ( $yClient/2$ ) to find the y position for the output. Also, the y value of the key bitmap receives another character. The application draws the KOALA bitmap using the BitBlt function.

The StretchBlt function enlarges the pie bitmap. When the application ends, both bitmaps and the timer must be removed. You could also remove other timers instead.

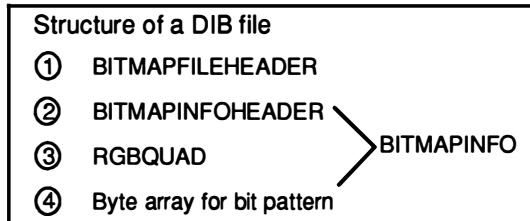
## Device-independent bitmaps (DIB)

Bitmaps that are dependent on devices can cause problems when graphics are transferred from one computer to another. Therefore, Windows also provides device-independent bitmaps. Two applications that use this format are PAINTBRUSH (PBRUSH.EXE) and SDKPAINT.EXE.

## DIB structure

PAINTBRUSH (PBRUSH.EXE) and SDKPAINT.EXE create device-independent bitmap files. These files consist of a data structure

called BITMAPFILEHEADER, followed by a BITMAPINFOHEADER structure and an array containing the bit pattern.



When you compare the structures of a device-dependent bitmap file and a DIB file, you'll see that the device-independent bitmap file contains more arrays.

All device-independent bitmap files begin with the letters "BM".

Old bitmap files (i.e., bitmaps created using the ICONEDIT.EXE application) can be converted to DIB files by loading them into SDKPAINT.EXE and saving them again.

SDKPAINT adds the necessary information. The variables (except for the bmWidthBytes array) can be read using the GetObject function. This function computes the value of this variable, in pixels, from the biWidth array and biBitCount variable.

### Structure of .BMP files

<i>'device-dependent'</i>		
BITMAPstructure	Data type	Value
bmType bmWidth bmHeight bmWidthBytes bmPlane bmBitsPixel	BYTE	2
	BYTE	80H for discardable bitmaps
	WORD	0
	WORD	Width in pixels
	WORD	Height in pixels
	WORD	Width in bytes
	BYTE	Number of 'color planes'
	BYTE	Number of color units per pixel
	WORD	0
	WORD	0
Array for bit pattern		
<i>'device-independent'</i>		
Field name	Data type	Value
① bfType bfSize bfReserve bfReserve bfOffBits biSize biWidth biHeight biPlanes biBitCount ② biStyle biSizeImage biXPelsPerMeter biYPelsPerMeter biClrUsed biClrImportant	WORD	BM
	DWORD	File size
	WORD	0
	WORD	0
	DWORD	Offset to bit pattern
	DWORD	Size of bit structure
	DWORD	Width in pixels
	DWORD	Height in pixels
	WORD	Number of color planes
	WORD	Number of color bits per pixel
③ rgBlue rgGreen rgRed rgReserved	DWORD	Type of compression
	DWORD	Size of compression bitmap
	DWORD	Horizontal pixel-resolution of device per meter
	DWORD	Vertical pixels-resolution of device per meter
④	DWORD	Number of indices in the color table
	DWORD	Number of important color indices
Array for the bit pattern		

The biBitCount variable indicates how many bits are needed to display a pixel, and acts as part of the bmiColors array because a two bit array can contain up to 256 entries. Two entries are sufficient for monochrome bitmaps.

If the bit is deleted from the bit pattern, the color of the first entry is passed. Otherwise, the second color is passed. Color bitmaps provide the maximum number of visible colors by using three different options.

biBitCount bits per pixel	Number of entries in bmiColors table	Description
1	2	Monochrome bitmap Bit=1-> Pixel uses color of 1st entry Bit=0-> Pixel uses color of 2nd entry
4	16	Bitmap with max. 16 colors. Each pixel appears through 4 bit index in bmiColors table.
8	256	Bitmap with max. 256 colors. Each pixel appears through 8 bit index in bmiColors table.
24	NULL	Bitmap with max. 16 Meg colors. Every three bytes in bitmap array represent red, blue and green intensities of one pixel.

The bmiColors array can be conveyed as an array of 16 bit integer values, displaying an index of the current logical color palette and organizing the RGB values. In this case, the DIB functions must send information to DIB\_PAL\_COLORS through the wUsage parameter.

In addition to the BITMAPINFO structure, you'll find a structure named BITMAPCOREINFO. This structure characterizes a device-independent bitmap in Presentation Manager (OS/2) format. This allows Windows to access this type of bitmap.

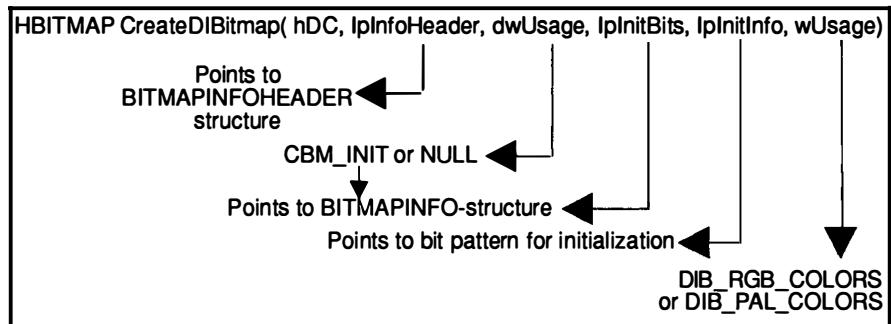
There is one other major difference between device-dependent and device-independent bitmaps. The point of origin for a device-independent bitmap lies in the lower-left corner of the graphic, instead of the upper-left corner used by device-dependent bitmaps.

## DIB functions

Windows 3.0 contains four functions that can control device-independent bitmaps.

CreateDIBitmap  
GetDIBits  
SetDIBits  
SetDIBitsToDevice

The CreateDIBitmap function uses a DIB specification (BITMAPINFO) to create a memory bitmap that has the proper size and device-dependent color format.



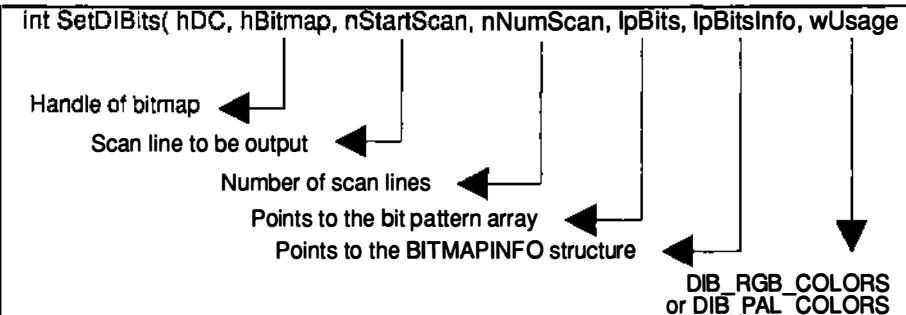
When the `dwUsage` parameter of the `CBM_INIT` value is set, the bitmap initializes it with the bits. These bits are addressed by a pointer in the last parameter.

This function converts the device-independent bitmap definition into a device-dependent format, from which the first parameter takes the device context.

The device-dependent bit pattern is then copied to memory, so that it can be accessed later through the handle received from memory.

If no bits are passed during initialization, the empty bitmap can still be written using the `SetDIBits` function. Also, an existing bitmap can be filled with a new bit pattern.

Therefore, the bitmap cannot be selected in the device context. The variables of the `BITMAPINFO` structure determine the bytes to be set, and how they should be interpreted.

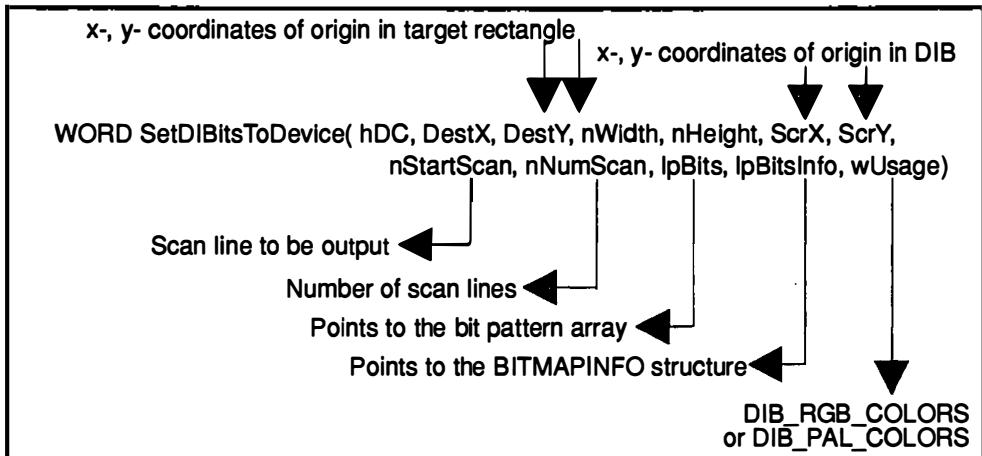


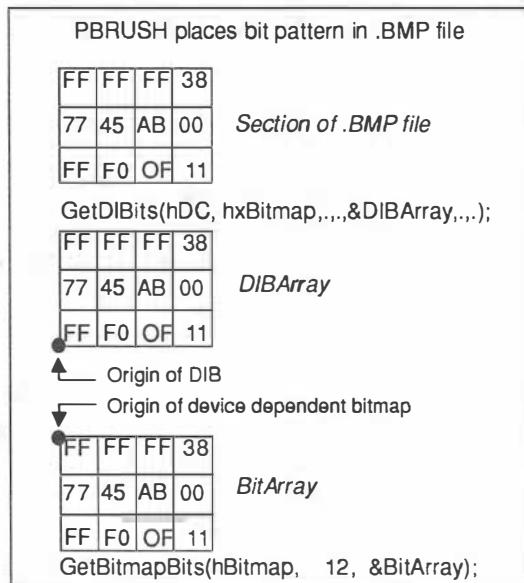
The `GetDIBits` function receives the same parameters as the `SetDIBits` function. It copies the bits from the pattern to a pointer specific array, in device-independent format. The return value depends on how many raster lines were transferred.

```
int GetDIBits( hDC, hBitmap, nStartScan, nNumScan, lpBits, lpBitsInfo, wUsage )
```

To output a device-independent bitmap, you must write the `SetDIBitsToDevice` to a device interface. A memory device context isn't needed. The bitmap is defined as a `BITMAPINFO` structure and a bit array (not as a handle).

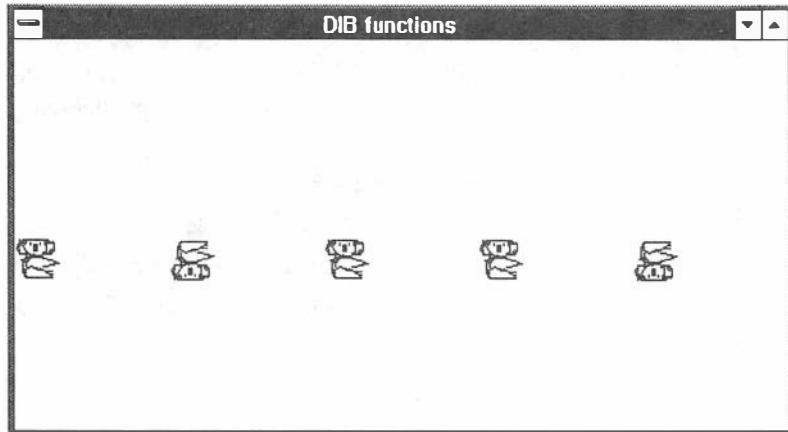
The `SrcX`, `SrcY`, `nWidth`, and `nHeight` parameters, and the `nStartScan`, `nNumScan` values send all or part of the total graphic for display.





If you work with monochrome device-independent bitmaps, the same functions used for device-dependent bitmaps should apply. However, since they use different reference points, you may need to mirror the graphics vertically. To do this, use the GetBitmapBits function to exchange the raster line sequence. You should use GetBitmapBits instead of GetDIBits when the bit pattern array is filled with data.

## Example of DIB functions



For this example, we used the monochrome KOALA bitmap from the previous example. We did this because a different series of raster characters are used in device-independent bitmap routines than in device-dependent bitmaps.

The graphic will be displayed twice by the SetDIBitsToDevice function and three times by the BitBlt function. Two arrays are used to convey the bitmap file with the help of the GetDIBits and GetBitmapBits functions.

#### New functions

	Brief description
CreateDIBitmap	Creates a bitmap based on DIB specification
GetBitmapBits	Copies bits from a bitmap to an array
GetDIBits	Copies bitmap bits in DIB format to an array
SetDIBitsToDevice	Displays a DIB directly

#### New structures

	Brief description
BITMAPINFO	DIB information, divided into: BITMAPINFOHEADER (DIB format information) RGBQUAD (RGB color structure)

## Source code: BIT2.C

```
/** BIT2.C ****
/* Displays the same bitmap in different ways */
****

#include "windows.h"                                // Include windows.h header file
#include "math.h"                                    // Include math.h header file
#include "string.h"                                  // Include string.h header file

BOOL Bit2Init ( HANDLE );
long FAR PASCAL Bit2WndProc( HWND, unsigned, WORD, LONG);

/** WinMain (main function for all Windows applications) *****

int PASCAL WinMain( hInstance, hPrevInstance, lpszCmdLine, cmdShow )
HANDLE hInstance, hPrevInstance;                  // Current & previous instance
LPSTR lpszCmdLine;                            // Long ptr - string after prg name
int cmdShow;                                  // App. window's appearance
{
```

## Bitmaps

---

```
MSG      msg;                      // Message handle
HWND     hWnd;                     // Window handle

if (!hPrevInstance)                // Initialize first instance
{
    if (!Bit2Init( hInstance ))
        return FALSE;
}

hWnd = CreateWindow("Bit2", "DIB functions",
    WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, 0, CW_USEDEFAULT, 0,
    NULL, NULL, hInstance, NULL);

ShowWindow( hWnd, cmdShow );
UpdateWindow( hWnd );

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

return (int)msg.wParam;
}

BOOL Bit2Init( hInstance )
HANDLE hInstance;
{
    WNDCLASS      wcBit2Class;

    wcBit2Class.hCursor      = LoadCursor( NULL, IDC_ARROW );
    wcBit2Class.hIcon        = LoadIcon( NULL, IDI_APPLICATION );
    wcBit2Class.lpszMenuName = NULL;
    wcBit2Class.lpszClassName = "Bit2";
    wcBit2Class.hbrBackground = GetStockObject( WHITE_BRUSH );
    wcBit2Class.hInstance     = hInstance;
    wcBit2Class.style         = CS_HREDRAW | CS_VREDRAW;
    wcBit2Class.lpfnWndProc   = Bit2WndProc;
    wcBit2Class.cbClsExtra    = 0 ;
    wcBit2Class.cbWndExtra    = 0 ;

    if (!RegisterClass( &wcBit2Class ) )
        return FALSE;

    return TRUE;
}
```

```
long FAR PASCAL Bit2WndProc( hWnd, message, wParam, lParam )
HWND      hWnd;
unsigned    message;
WORD       wParam;
LONG       lParam;
{
    HANDLE   hInst;
    PAINTSTRUCT      ps;
    HDC        hDC, hMem;
    static HBITMAP    hBitmap, hBit, hDIB, hMixed, hOld;
    static BITMAP     bm;
    static BITMAPINFO DIBInfo;
    static BYTE       DIBArray[200], BitArray[200];
    static short      xClient, yClient;

    switch (message)
    {
        case WM_CREATE:
            hInst = GetWindowWord( hWnd, GWW_HINSTANCE);
            hBitmap = LoadBitmap(hInst, "koala");
            GetObject(hBitmap, sizeof(BITMAP), (LPSTR)&bm);

            memset(&DIBInfo.bmiHeader, 0, sizeof(BITMAPINFOHEADER));
            DIBInfo.bmiHeader.biSize = sizeof(BITMAPINFOHEADER);
            DIBInfo.bmiHeader.biWidth = bm.bmWidth;
            DIBInfo.bmiHeader.biHeight = bm.bmHeight;
            DIBInfo.bmiHeader.biPlanes = 1;
            DIBInfo.bmiHeader.biBitCount = 1;

            DIBInfo.bmiColors[0].rgbRed = 0;
            DIBInfo.bmiColors[0].rgbGreen = 0;
            DIBInfo.bmiColors[0].rgbBlue = 0;
            DIBInfo.bmiColors[0].rgbReserved = 0;
            DIBInfo.bmiColors[1].rgbRed = 0xff;
            DIBInfo.bmiColors[1].rgbGreen = 0xff;
            DIBInfo.bmiColors[1].rgbBlue = 0xff;
            DIBInfo.bmiColors[1].rgbReserved = 0;

            hDC = GetDC( hWnd);
            // Handle bitmap as DIB: Lower-left origin
            GetDIBits(hDC, hBitmap, 0, bm.bmHeight, DIBArray, &DIBInfo,
DIB_RGB_COLORS);

            // Create device-dependent bitmap
            hDIB = CreateDIBitmap(hDC, &DIBInfo.bmiHeader, CBM_INIT,
DIBArray,
```

## Bitmaps

---

```
    &DIBInfo, DIB_RGB_COLORS);
    ReleaseDC(hWnd, hDC);

    hMixed = CreateBitmap(32,32,1,1,(LPSTR)DIBArray);

    // Handle bitmap as DIB: upper-left origin
    GetBitmapBits(hBitmap, bm.bmHeight*bm.bmWidthBytes,
BitArray);

    // Create device-dependent bitmap
    hBit = CreateBitmap(32,32,1,1,(LPSTR)BitArray);
    break;

    case WM_SIZE:
        xClient = LOWORD(lParam);
        yClient = HIWORD(lParam);
        break;

    case WM_PAINT:
        hDC = BeginPaint(hWnd, &ps);

        SetDIBitsToDevice(hDC, 0,yClient/2,
bm.bmWidth,bm.bmHeight,0,0,0,32, DIBArray,
&DIBInfo, DIB_RGB_COLORS);

        // Stand koala on its head
        SetDIBitsToDevice(hDC, 1*xClient/5,yClient/2,
bm.bmWidth,bm.bmHeight,
0,0,0,32, BitArray, &DIBInfo, DIB_RGB_COLORS);

        hMem = CreateCompatibleDC(hDC);
        hOld = SelectObject(hMem, hBit);
        BitBlt(hDC, 2*xClient/5,yClient/2, 32,32,hMem,0,0,SRCCOPY);

        SelectObject(hMem, hDIB);
        BitBlt(hDC, 3*xClient/5,yClient/2, 32,32,hMem,0,0,SRCCOPY);

        SelectObject(hMem, hMixed);
    // Stand koala on its head
        BitBlt(hDC, 4*xClient/5,yClient/2, 32,32,hMem,0,0,SRCCOPY);
        SelectObject(hMem, hOld);
        DeleteDC(hMem);

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

```

        case WM_DESTROY:           // Destroy window
            DeleteObject(hBitmap);
            DeleteObject(hBit);
            DeleteObject(hDIB);
            DeleteObject(hMixed);
            PostQuitMessage(0);
            break;

        default:
            return (DefWindowProc( hWnd, message, wParam, lParam ));
            break;
    }
    return(0L);
}

```

## Module definition file: BIT2.DEF

```

NAME      Bit2

DESCRIPTION 'Device-independent bitmap functions'

EXETYPE   WINDOWS

STUB      'WINSTUB.EXE'

CODE      PRELOAD MOVEABLE
DATA      PRELOAD MOVEABLE MULTIPLE

HEAPSIZE  4096
STACKSIZE 4096

EXPORTS   Bit2WndProc

```

## Resource script: BIT2.RC

```
koala      BITMAP      koala.bmp
```

To compile and link this application, use the RCOMPILE.BAT batch file described earlier in this book. Here's the listing again:

```

cl -c -Gw -Zp %1
rc -r %1.rc
link /align:16 %1,%1.exe,,libw+slibcew,%1.def
rc %1.res

```

Save this file to a directory contained in your path or one containing the source, module definition and resource scripts. Type the following and press **Enter** to compile BIT2:

```
rcompile bit2
```

## How BIT2.EXE works

This example presents five different combinations of device-dependent and device-independent bitmap functions.

The WM\_CREATE message fills the DIBInfo variable of the BITMAPINFO structure with the following DIB functions. The values are read from the KOALA.BMP file by the .RC file. The bmiColors color table contains two entries, which ensures that the bitmap is handled as a monochrome bitmap.

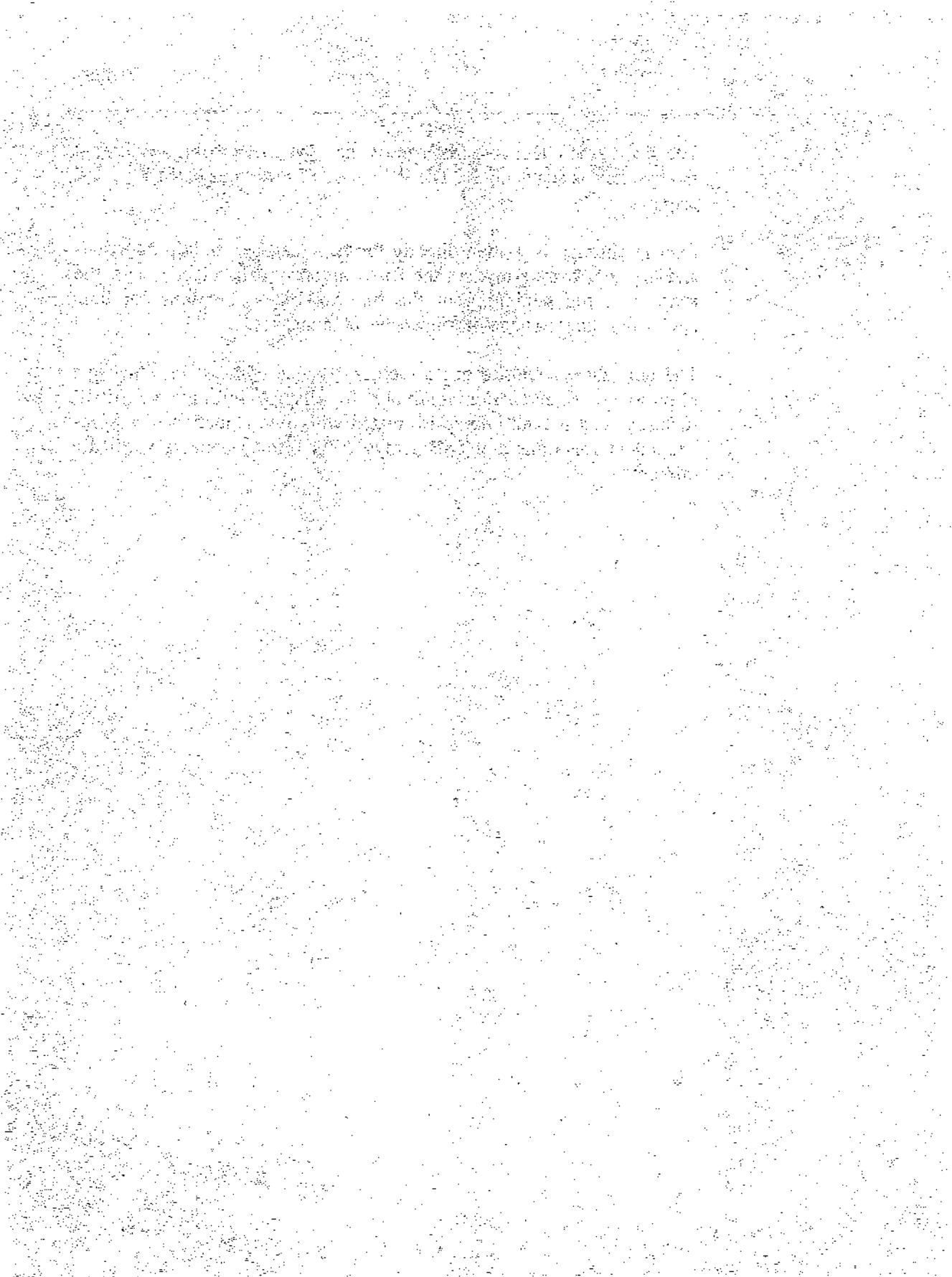
Then the bits specified by the bit pattern are copied to the BitArray array by the GetDIBits function, the DIBArray array, and the GetBitmapBits device-dependent function. The CreateDIBitmap function is called twice, and accesses one of the two arrays. CreateDIBitmap creates a third bitmap.

<p>device-independent Origin: links lower left</p> <p>GetDIBits(...,...,&amp; <i>DIBARRAY</i> ,...)</p>	<p>device-dependent Origin: links lower left</p> <p>GetBitmapBits(...,&amp; <i>BitARRAY</i>)</p>
<p>① <i>DIBArray</i> SetDIBitsToDevice(...)</p> <p>② <i>BitArray</i> SetDIBitsToDevice(...)</p> <p>③ CreateDIBitmap(...,...,&amp; <i>BitArray</i> ) BitBlt(...)</p> <p>④ CreateDIBitmap(...,...,&amp; <i>BitArray</i>,...) BitBlt(...)</p> <p>⑤ CreateDIBitmap(...,...,&amp; <i>BitArray</i> ) =&gt; Bitmap is reflected vertically BitBlt(...)</p>	<p>=&gt; Bitmap is reflected vertically</p>

The size of the client area (defined by the WM\_SIZE message) displays the five koalas sequentially. The WM\_PAINT message controls the five displays.

First the bitmap is written directly to the screen by the DIB functions and the DIBArray array. As the BitArray array calls other arrays, the graphic at the beginning of the line combines functions for both device-dependent and device-independent bitmaps.

The next three graphics use a memory device context, in which the bitmaps are selected consecutively, for display through the BitBlt function. The second bitmap is created with a combination of device-dependent functions and DIB arrays. This bitmap appears vertically mirrored.



# Scroll Bars

## Overview

The movement of data in the client area or in other areas of a window is referred to as scrolling. By scrolling, you can display text or graphics that aren't currently visible. For example, if a document's margins are wider than the window displaying it, scrolling enables you to see the entire text.

Scroll bars are needed in order to scroll the contents of a window. These bars can either be stand-alone controls displayed in a child window or defined as style parameters when creating a window. If the scroll bars are defined as style parameters, they can be placed in the right and bottom borders of the client area. We'll discuss scroll bars in detail in this chapter.

## Working with scroll bars

Scroll bars were designed to be user-friendly. The user changes position in the client area by dragging the thumb (the square button indicating the document's position) or clicking on the scroll arrows. Once this is done, the application must respond by displaying the appropriate text or graphics.

## Defining scroll bars

The third parameter found in the CreateWindow function specifies the window's style. A standard window frequently has multiple flags set, such as WS\_OVERLAPPEDWINDOW. The WS\_HSCROLL parameter specifies a horizontal scroll bar and the WS\_VSCROLL parameter specifies a vertical scroll bar. Both of these parameters are implemented with the help of OR operators.

```
hWnd = CreateWindow( "Scroll", "Scroll Bar Application", WS_OVERLAPPEDWINDOW |  
WSHSCROLL | WS_VSCROLL, CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance,  
NULL );
```

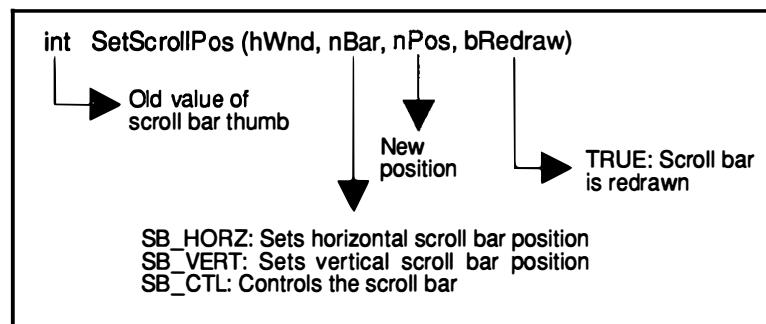
A screen driver maintains the width of the vertical scroll bar and the height of the horizontal scroll bar. The values for width and height can

be sent to the GetSystemMetrics function through the SM\_CYHSCROLL and SM\_CXVSCROLL parameters.

## Scroll range and scroll bar position

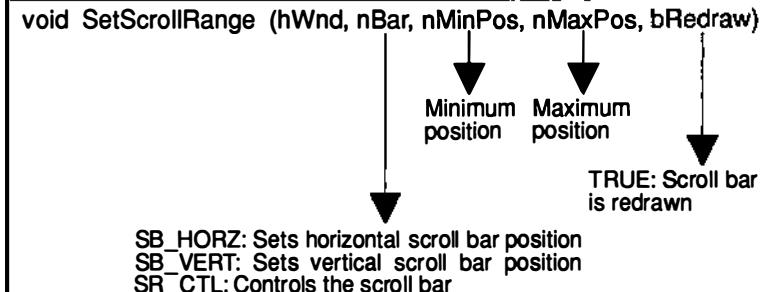
Every scroll bar contains a scroll range and a current position. The range is assigned minimum and maximum values. The minimum value represents the top (vertical) or left (horizontal) end of the scroll bar, while the maximum value represents the bottom (vertical) or right (horizontal) end of the scroll bar.

The thumb (the square within the scroll bar) indicates the current position in the file. The programmer is responsible for making certain that the thumb indicates the correct position in the document.



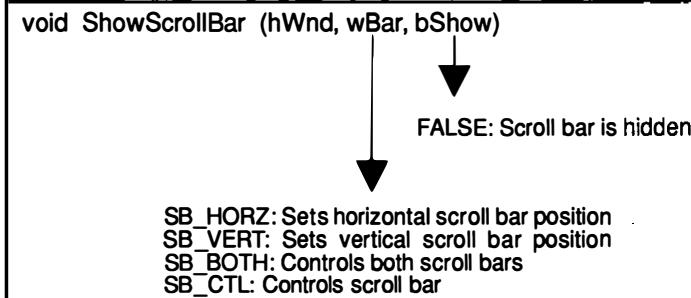
The thumb can be assigned any integer value that applies to the scroll bar. This value places the thumb at that position within the scroll bar. If the thumb is assigned a value that is outside the defined scroll range, the system will automatically default to the minimum or maximum value.

The `SetScrollRange` function initializes the spread of the scroll range. Both corner values can be any integer values if the minimum value is less than the maximum value.



The size of the scroll range can also be determined by the data being displayed. For example, if you display a table in the client area consisting of 80 rows and 10 columns, the vertical scroll range values can be from 1 to 80. The horizontal scroll range values can be from 1 to 10 (or 0 to 9). These values can be changed at any time.

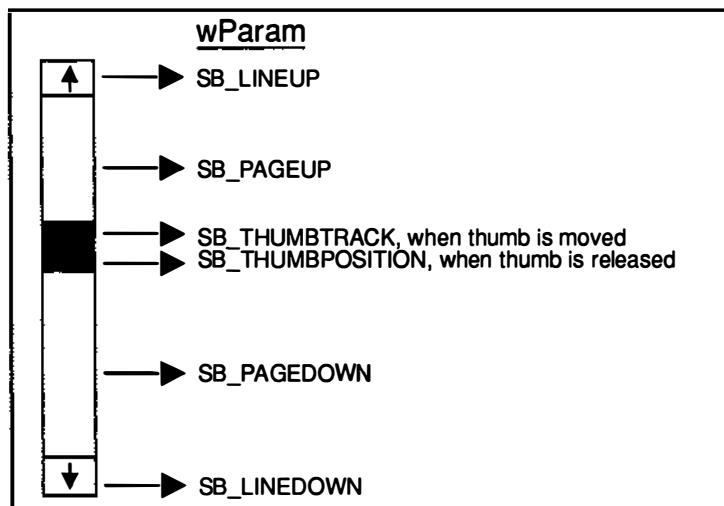
If both corner points contain the same values, the scroll bar is deactivated (removed from the window). This also occurs when the programmer places the value FALSE in the third parameter of the `ShowScrollBar` function.



This function can also be used with the first method of scroll bar control design, which we mentioned at the beginning of this chapter.

## Scroll bar messages

When the user clicks anywhere within a scroll bar or drags the thumb, Windows sends a `WM_HSCROLL` or `WM_VSCROLL` message to the window routine. The `wParam` parameter specifies the exact action that occurred:



The wParam parameter contains three messages in addition to the six mentioned above:

SB_BOTTOM	Thumb is in upper (left) end
SB_TOP	Thumb is in lower (right) end
SB_ENDSCROLL	Mouse button released

The SB\_ENDSCROLL parameter is always sent when the user releases the pressed mouse button. The SB\_TOP and SB\_BOTTOM messages indicate that the thumb is located at the minimum or maximum position within the scroll bar. (This applies to scroll bar controls only, instead of the standard scroll bar.)

The WM\_VSCROLL and WM\_HSCROLL messages are sent when the programmer has included keyboard support for scroll bars.

The SB\_THUMBTRACK and SB\_THUMBPOSITION messages indicate the thumb's current position, and return this information to the low word of lParam. This position lies between both corner positions defining the scroll bar range.

When the user drags the thumb with the mouse, wParam uses the SB\_THUMBTRACK message. When the user releases the mouse button, wParam passes the SB\_THUMBPOSITION message. This can be

difficult to program because the application must compare certain factors and then select the appropriate value. For example:

The amount of data to be displayed at one time.

The complexity of the data (text or graphics).

The speed at which the user drags the thumb.

Sometimes processing the SB\_THUMBTRACK message can lead to problems when preparing data for display. However, the thumb gives the user a good indication of the document or graphic's position within the window.

## Keyboard support

Unlike the scroll bar controls, the standard scroll bars offer automatic keyboard support. When the user presses the specified keys, the thumb moves. The programmer must specify the keys needed for this movement with the WM\_KEYDOWN message.

The wParam parameter sends the virtual key code of the key pressed. Most of the keys defined in the WINDOWS.H include file can be used for controlling the scroll bars:

Virtual key code	Key	Movement
VK_LEFT	Left arrow	One page or column left
VK_RIGHT	Right arrow	One page or column right
VK_UP	Up arrow	One line up
VK_DOWN	Down arrow	One line down
VK_NEXT	PgDn	One page up
VK_PRIOR	PgUp	One page down
VK_END	End	End
VK_HOME	Home	Beginning

When the user presses a key, the SendMessage function sends either a WM\_HSCROLL message or a WM\_VSCROLL message to the corresponding window routine's wParam. This controls the scroll bar in the same way that the mouse can be used to control the scroll bar.

```
case WM_KEYDOWN:
    switch (wParam)
    {
        case VK_UP:
            SendMessage (hWnd, WM_VSCROLL, SB_LINEUP, 0L);
```

```
        break;

    case VK_DOWN:
        SendMessage (hWnd, WM_VSCROLL, SB_LINEDOWN, 0L);
        break;

    case VK_LEFT:
        SendMessage (hWnd, WM_HSCROLL, SB_LINEUP, 0L);
        break;

    .
    .

}
```

## Scrolling

The programmer must plan for the thumb's placement in its new position and how the scrolled data will be displayed. The easiest way to do this is by calling the InvalidateRect function each time a scroll bar message occurs but invalidates the entire client area. The WM\_PAINT message then tries to display the entire document. Windows can display the clipping rectangle only in the client area.

This method unfortunately causes flickering in the client area each time the area is changed by a scroll bar. This occurs because the contents of the client area must be deleted and redrawn.

The ScrollWindow function can eliminate flickering during scrolling. This function moves a section of the client area a specific distance without deleting the section. The rest of the client area is marked as invalid because it wasn't needed at the time. This range can then be provided by the next WM\_PAINT message. ScrollWindow uses the BitBlt function to move a predetermined segment of the output within the client area.

```
void ScrollWindow( hWnd, xAmount, yAmount, lpRect, lpClipRect )
```

Parameter	Type	Brief description
hWnd	HWND	Handle of the window whose client area will be scrolled.
xAmount	int	Scroll length in x direction (in Device Units).
yAmount	int	Scroll length in y direction (in Device Units).
lpRect	LPRECT	Pointer to rectangle to be scrolled. If NULL, entire client area scrolls.
lpClipRect	LPRECT	Pointer to rectangle appointed as clipping rectangle. If NULL, entire client area scrolls.

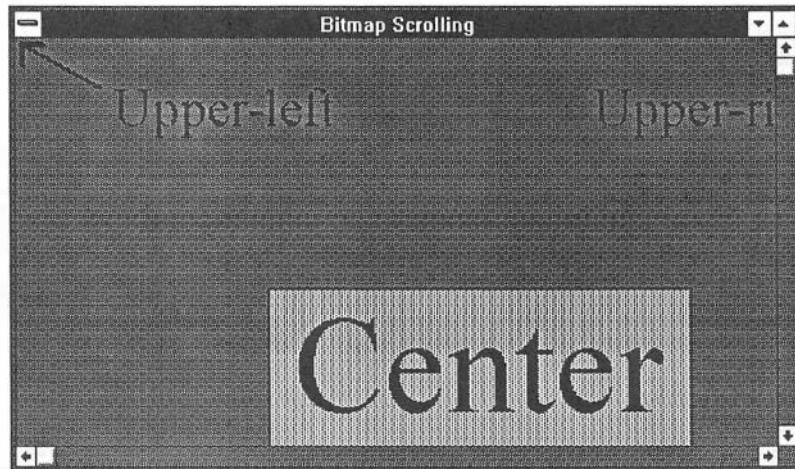
The xAmount and yAmount parameters contain positive values when scrolling down (vertical) or right (horizontal) occurs.

Since only a portion of the client area can be scrolled, selected objects (e.g., a child window) can also change position.

The ScrollWindow function has been rendered invalid. So the coordinates of a section of the client area can be passed to the WM\_PAINT message from the ps variable found in the PAINTSTRUCT structure. Once this has happened, this range can be used for a new output.

The WM\_PAINT message can do this through the UpdateWindow function, which is called through ScrollWindow. The thumb movement and data scrolling occurs at approximately the same time. This happens because instead of placing the message in the application queue, UpdateWindow sends the message directly to the window function.

## Example of scroll bars



This application uses vertical and horizontal scroll bars. It displays the ABACUS.BMP bitmap included on the companion diskette for this book. You can view the entire graphic by scrolling around the window with the scroll bars.

### New messages

WM\_HSCROLL  
WM\_VSCROLL

### New functions

GetScrollPos  
SetScrollPos  
SetScrollRange  
ScrollWindow

### New macros

max  
min

### Brief description

Indicates horizontal scroll bar was clicked  
Indicates vertical scroll bar was clicked

### Brief description

Receives the current thumb position  
Sets the thumb to a new position  
Sets the range of the scroll bar  
Scrolls the contents of the client area

### Brief description

Returns the two greatest values  
Returns the two smallest values

## Source code: SCROLL.C

```
/** SCROLL.C ****
/** Demonstrates a scrolling window using a bitmap ****
/****

#include "windows.h"                                // Include windows.h header file

long FAR PASCAL ScrollWndProc (HWND, unsigned, WORD, LONG) ;

/** WinMain (main window function for all windows applications ****

int PASCAL WinMain (hInstance, hPrevInstance, lpszCmdLine, nCmdShow)
    HANDLE hInstance, hPrevInstance ;      // Current & previous instance
    LPSTR lpszCmdLine ;                  // Long ptr - string after prg name
    int     nCmdShow ;                  // App. window's appearance
{
    HWND    hWnd ;                      // Window handle
    MSG     msg ;                      // Message handle

/** Specify window class ****

WNDCLASS      wndclass ;

    if (!hPrevInstance)
    {
        wndclass.style      = CS_HREDRAW | CS_VREDRAW ;
        wndclass.lpfnWndProc = ScrollWndProc;
        wndclass.cbClsExtra = 0 ;
        wndclass.cbWndExtra = 0 ;
        wndclass.hInstance   = hInstance ;
        wndclass.hIcon       = LoadIcon (NULL, IDI_APPLICATION) ;
        wndclass.hCursor     = LoadCursor (NULL, IDC_ARROW) ;
        wndclass.hbrBackground = GetStockObject (WHITE_BRUSH) ;
        wndclass.lpszMenuName = NULL ;
        wndclass.lpszClassName = "Scroll" ;

        if (!RegisterClass (&wndclass))
            return FALSE ;
    }

/** Specify appearance of application's main window ****

hWnd = CreateWindow ("Scroll", " Bitmap Scrolling",
    WS_OVERLAPPEDWINDOW ! WS_VSCROLL | WS_HSCROLL,
    CW_USEDEFAULT, 0,
    CW_USEDEFAULT, 0,
```

## Scroll Bars

---

```
        NULL,
        NULL,
        hInstance,
        NULL) ;

ShowWindow (hWnd, nCmdShow) ;           // Make window visible
UpdateWindow (hWnd) ;                 // Update window

while (GetMessage (&msg, NULL, 0, 0))// Message reading
{
    TranslateMessage (&msg) ;      // Message translation
    DispatchMessage (&msg) ;     // Send message to Windows
}
return msg.wParam ;
}

/** ScrollWndProc *****/
/** Main window function: All messages are sent to this window      */
/** *****/

long FAR PASCAL ScrollWndProc (hWnd, message, wParam, lParam)
    HWND      hWnd ;
    unsigned message ;
    WORD      wParam ;
    LONG      lParam ;
{
    HANDLE   hInst;
    static short  xClient, yClient;
    static short  sVertPos, sVertOld, sHorzPos, sHorzOld;
    static BITMAP  bm;
    static HBITMAP hBitmap, hBitOld;
    static short   sMaxHeight, sMaxWidth;
    HDC       hDC, hMemDC;
    PAINTSTRUCT ps;

    switch (message)
    {
        case WM_CREATE:
            hInst = GetWindowWord(hWnd, GWW_HINSTANCE);
            hBitmap = LoadBitmap(hInst, "Abacus");
            GetObject(hBitmap, sizeof(BITMAP), (LPSTR)&bm);
            sMaxHeight = bm.bmHeight ;
            sMaxWidth = bm.bmWidth;

        /** Set scroll bar limits *****/
            SetScrollRange (hWnd, SB_VERT, 0, sMaxHeight, FALSE) ;
            SetScrollPos   (hWnd, SB_VERT, sVertPos, TRUE) ;
    }
}
```

```
    SetScrollRange (hWnd, SB_HORZ, 0, sMaxWidth, FALSE) ;
    SetScrollPos   (hWnd, SB_HORZ, sHorzPos, TRUE) ;
    break ;

case WM_SIZE:           // Client area size
    yClient = HIWORD (lParam) ;
    xClient = LOWORD (lParam) ;
    break ;

case WM_VSCROLL:        // Vertical scroll
    sVertOld = sVertPos;

    switch (wParam)
    {
        case SB_LINEUP:
            sVertPos -= 1 ;
            break ;

        case SB_LINEDOWN:
            sVertPos += 1 ;
            break ;

        case SB_PAGEUP:
            sVertPos -= yClient ;
            break ;

        case SB_PAGEDOWN:
            sVertPos += yClient ;
            break ;

        case SB_TOP:
            sVertPos = 0;
            break;

        case SB_BOTTOM:
            sVertPos = sMaxHeight;
            break;

        case SB_THUMBTRACK:
            sVertPos = LOWORD(lParam);
            break ;

        default:
            break ;
    }
sVertPos = max (0, min (sVertPos, sMaxHeight)) ;
```

```
    if (sVertPos != GetScrollPos (hWnd, SB_VERT))
    {
        ScrollWindow (hWnd, 0,sVertOld-sVertPos,
                      NULL, NULL);
        UpdateWindow(hWnd);
        SetScrollPos (hWnd, SB_VERT, sVertPos, TRUE) ;
    }
    break ;

case WM_HSCROLL:           // Horizontal scroll
    sHorzOld = sHorzPos;

    switch (wParam)
    {
        case SB_LINEUP:
            sHorzPos -= 1 ;
            break ;

        case SB_LINEDOWN:
            sHorzPos += 1 ;
            break ;

        case SB_PAGEUP:
            sHorzPos -= xClient ;
            break ;

        case SB_PAGEDOWN:
            sHorzPos += xClient ;
            break ;

        case SB_THUMBUPOFFSET:
            sHorzPos = LOWORD(lParam);
            break ;

        default:
            break;
    }

    sHorzPos = max (0, min (sHorzPos, sMaxWidth)) ;

    if (sHorzPos != GetScrollPos (hWnd, SB_HORZ))
    {
        ScrollWindow (hWnd, sHorzOld-sHorzPos,0,
                      NULL, NULL);
        UpdateWindow(hWnd);
        SetScrollPos (hWnd, SB_HORZ, sHorzPos, TRUE) ;
    }
}
```

```
        break ;

case WM_KEYDOWN:           // Check mouse
    switch (wParam)
    {
        case VK_UP:
            SendMessage (hWnd, WM_VSCROLL,
                          SB_LINEUP, 0L);
            break;

        case VK_DOWN:
            SendMessage (hWnd, WM_VSCROLL,
                          SB_LINEDOWN, 0L);
            break;

        case VK_PRIOR:
            SendMessage (hWnd, WM_VSCROLL,
                          SB_PAGEUP, 0L);
            break;

        case VK_NEXT:
            SendMessage (hWnd, WM_VSCROLL,
                          SB_PAGEDOWN, 0L);
            break;

        case VK_LEFT:
            SendMessage (hWnd, WM_HSCROLL,
                          SB_LINEUP, 0L);
            break;

        case VK_RIGHT:
            SendMessage (hWnd, WM_HSCROLL,
                          SB_LINEDOWN, 0L);
            break;

        case VK_HOME:
            SendMessage (hWnd, WM_VSCROLL,
                          SB_TOP, 0L);
            break;

        case VK_END:
            SendMessage (hWnd, WM_VSCROLL,
                          SB_BOTTOM, 0L);
            break;

        default:
            break;
    }
```

```
        break;

    case WM_PAINT:           // Client area redraw
        hDC = BeginPaint (hWnd, &ps) ;
        hMemDC = CreateCompatibleDC( hDC );
        hBitOld = SelectObject(hMemDC,hBitmap);
        BitBlt(hDC, ps.rcPaint.left, ps.rcPaint.top,
                ps.rcPaint.right,ps.rcPaint.bottom, hMemDC,
                sHorzPos+ps.rcPaint.left,
                sVertPos+ps.rcPaint.top, SRCCOPY);
        SelectObject (hMemDC,hBitOld);
        DeleteDC(hMemDC);
        EndPaint (hWnd, &ps) ;
        break ;

    case WM_DESTROY:          // Destroy window
        PostQuitMessage (0) ;
        break ;

    default:
        return DefWindowProc (hWnd, message, wParam, lParam) ;
    }
    return 0L ;
}
```

## Module definition file: SCROLL.DEF

NAME	Scroll
DESCRIPTION	'Scroll demonstration with bitmap'
EXETYPE	Windows
STUB	'WINSTUB.EXE'
CODE	PRELOAD MOVEABLE
DATA	PRELOAD MOVEABLE MULTIPLE
HEAPSIZE	4096
STACKSIZE	4096
EXPORTS	ScrollWndProc

## Resource script: SCROLL.RC

Abacus	BITMAP	abacus.bmp
--------	--------	------------

To compile and link this application, use the RCOMPILE.BAT batch file described earlier in this book. Here's the listing again:

```
cl -c -Gw -zp %1  
rc -r %1.rc  
link /align:16 %1,%1.exe,,libw+slibcew,%1.def  
rc %1.res
```

Save this file to a directory contained in your path or the directory containing your source, module definition and resource scripts. Type the following and press **Enter** to compile SCROLL:

```
rcompile scroll
```

## How SCROLL.EXE works

The .RC file indicates the bitmap name. You'll find the bitmap on the accompanying companion diskette or you can substitute another bitmap for the ABACUS bitmap. The WM\_CREATE message loads this file into the window and the GetWindowWord function provides the required instance handle.

The GetObject function determines the bitmap's height and width, from which the application sets the maximum values needed for the scroll bars. The minimum value for each scroll bar is zero. The thumbs appear at the far left (horizontal) and top (vertical) of the scroll bars.

For page-by-page scrolling, the current client area size is passed to the WM\_SIZE message from the lParam parameter.

The WM\_VSCROLL message takes the old thumb position because the ScrollWindow function will still need this information later. If the user scrolls the client area contents upward, the position value of the thumb must be decreased.

Therefore, the contents of the sVertPos variable must either be one or the size of the client area must be removed when the wParam parameter is either SB\_LINEUP or SB\_PAGEUP.

If the user scrolls the client area contents downward, the position value of the thumb must be increased. This is the responsibility of the SB\_LINEDOWN and SB\_PAGEDOWN messages and the sVertPos variable.

Vertical and horizontal scroll bars move the thumb differently. Also, initially vertical scroll bars don't react to the released mouse button. However, we won't demonstrate this here. The low word of the IParam parameter indicates the current position of the thumb. The SB\_TOP and SB\_BOTTOM values add keyboard support.

Regardless of the information in wParam, the sVertPos variable must be tested to make certain that its value is within the scroll range. If this value is too large or too small, the min and max macros set the minimum and maximum values. The max macro returns the largest parameter and the min macro returns the smallest parameter.

When the thumb's position changes from its most recent position, the ScrollWindow function moves the client area. To determine the amount of movement, find the difference between the old and new positions. The movement and redraw occurs at approximately the same time that the application calls the UpdateWindow function.

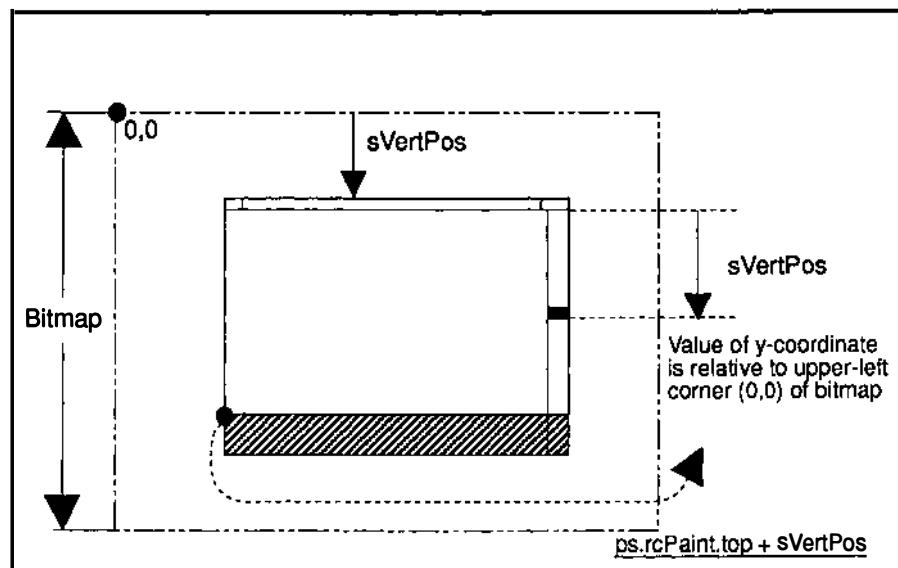
After the user moves the thumb, the thumb's new position must be set in the application. In our example, we compare the old and new positions with the GetScrollPos function, which returns the current position of the thumb. You could also compare the contents of the sVertPos and sVertOld variables.

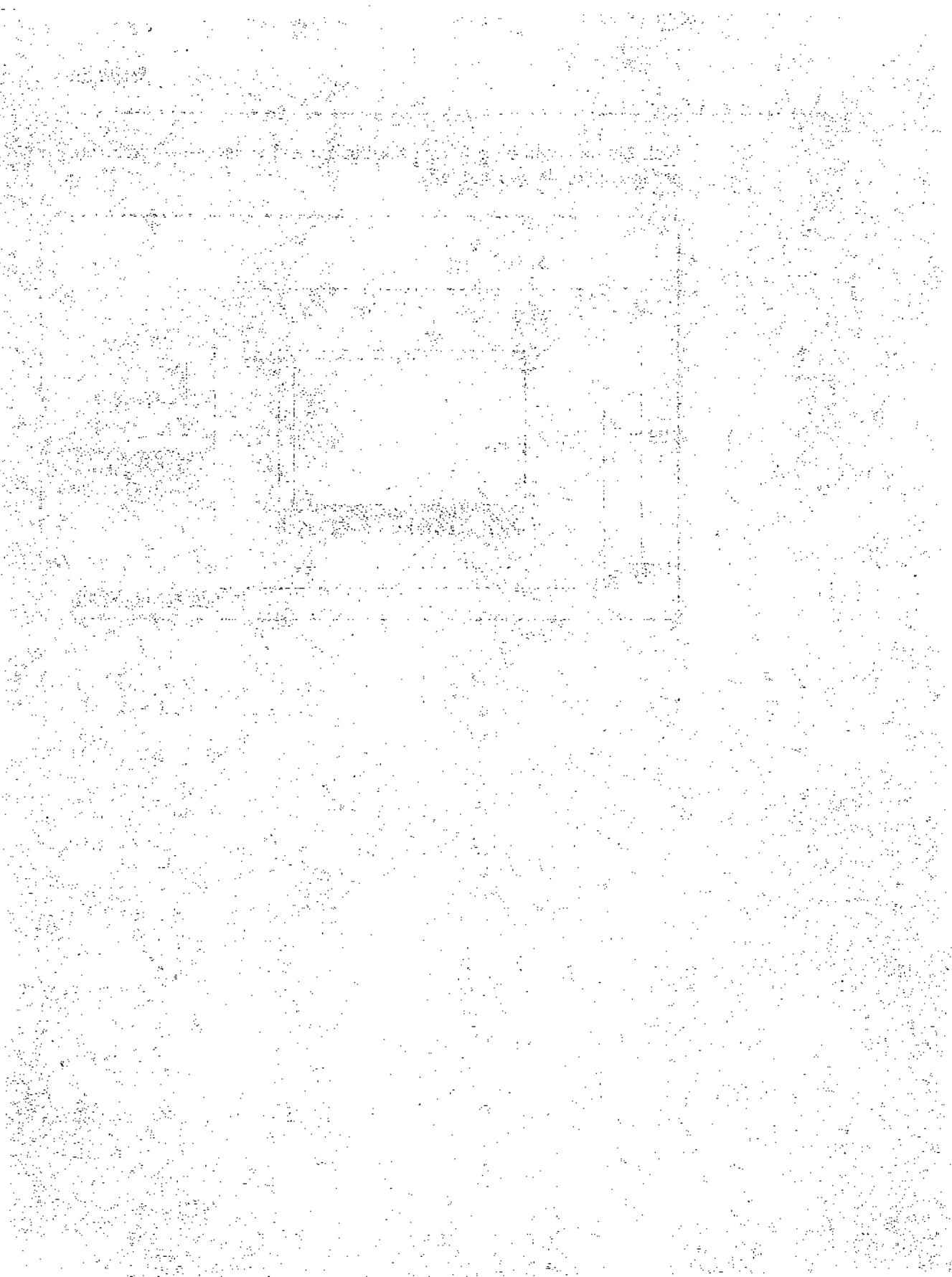
If the user clicks on the horizontal scroll bar or drags its thumb, the WM\_HSCROLL message occurs. The principles of movement here are similar to those cited above in the WM\_VSCROLL message, except that horizontal movement is based on x values instead of y values. The WM\_HSCROLL message reads whether the mouse button is released first, rather than the thumb itself. The new position can be found in IParam.

The new position of the bitmap display is calculated from the new thumb position and the upper-left corner of the invalid range of the client area. The following illustration demonstrates this, assuming that the bitmap scrolls up (i.e., the thumb is now in a lower position on the scroll bar).

There are several ways this application can be improved. For example, the scroll bars should disappear when the user makes the window the same size as (or larger than) the bitmap. Also, a different location for

each thumb could be set (e.g., the bottom and right end of each scroll bar instead of the top and left).





# The Clipboard

## Overview

The clipboard allows you to exchange information between applications. It contains an area of memory that is accessible to any Windows application that handles files. The clipboard is used for storing and passing handles to formatted data.

A handle returns file information, such as memory allocation within Windows. Frequently a handle will be passed to the clipboard when the Cut or Copy menu item is selected. Then the stored data can be placed in another file by selecting the Paste menu item. The CLIPBRD.EXE application is the visible contents of the clipboard, instead of the clipboard itself.

Several Windows functions apply to the clipboard:

Function	Description
ChangeClipboardChain	Viewer taken from viewer chain
CloseClipboard	Close clipboard
EmptyClipboard	Empty contents of clipboard
EnumClipboardFormats	Count number of clipboard formats
GetClipboardData	Get data from clipboard
GetClipboardFormatName	Get current clipboard format
GetClipboardOwner	Get clipboard owner's window handle
GetClipboardViewer	Get handle to first window in viewer chain
GetPriorityClipboardFormat	Get highest priority data format from Clipboard
IsClipboardFormatAvailable	Return TRUE if data is in specified format
OpenClipboard	Open clipboard
SetClipboardData	Copy a data handle to the Clipboard
SetClipboardViewer	Set handle in the viewer chain

The clipboard provides different formats for passing data. The following table lists the five most frequently used formats:

Format	Contents
CF_TEXT	Global memory contains a text that ends with a '0'
CF_BITMAP	A bitmap, whose handle is passed
CF_DIF	Global memory contains data in Data Interchange Format (DIF)
CF_METAFILEPICT	Global memory contains a Metafile picture structure
CF_SYLK	Global memory contains data in Symbolic Link format

The format must be passed when an application takes data from the clipboard with the GetClipboardData function. This format must match the format of the data stored in the clipboard. Otherwise, the data cannot be passed.

## Text format

The text format (CF\_TEXT) handles unformatted text, which is text that doesn't contain any control characters or format instructions. Unformatted text contains only ASCII (or ANSI in Windows) characters.

## Copying text to the clipboard

The following steps are needed to copy text to the clipboard:

### Copying text to global memory:

```
hClipData = GlobalAlloc(GMEM_MOVEABLE, GlobalSize(hMyData));
lpszClipData = GlobalLock(hClipData);
lpszMyData = GlobalLock(hMyData);
Istrcopy (lpszClipData, lpszMyData);
GlobalUnlock(hMyData);
GlobalUnlock(hClipData);
```

### Open the clipboard:

```
OpenClipboard(hWnd);
```

### Empty the clipboard's contents:

```
EmptyClipboard();
```

### Pass handle to global memory range:

```
SetClipboardData (CF_TEXT, hClipData);
```

## **Close the clipboard:**

```
CloseClipboard();
```

Whenever text is exchanged through the clipboard, it must be placed in a global memory area. This example assumes that the hMyData handle points to a global memory area in which the text can be found when the clipboard needs it. If the application is no longer using the handle given to the clipboard, the text is copied to another global memory area and its handle is written to the clipboard.

Therefore, a new global memory area is allocated, and the area is locked as exclusive by the GlobalLock function. The two resulting pointers, which represent a 32 bit address, are used by the lstrcpy function to copy the text into the new area. This releases the memory blocks.

Now we can open the clipboard and remove the old contents. When the window containing the clipboard has been opened, this data can be transferred. If the clipboard is empty, hClipData passes the CF\_TEXT format. Now the clipboard can be closed.

The OpenClipboard and CloseClipboard functions must be called from within the same message loop. As long as the clipboard is open, no other application can be opened or use the clipboard.

The clipboard should not give a handle to a memory block because it's been marked as allocated. Once this handle has been written to the clipboard, it can no longer access information from other applications because the clipboard is the owner of the handle.

This memory block isn't released when the application ends. Instead, it becomes the task of Windows, and remains until the EmptyClipboard function is called again.

## **Obtaining text from the clipboard**

### **Open the clipboard:**

```
OpenClipboard(hWnd);
```

## **Get handle to global memory area from clipboard:**

```
hClipBit = GetClipboardData(CF_TEXT);
```

## **Copy data to a global memory area:**

```
hMyData = GlobalAlloc(GMEM_MOVEABLE, GlobalSize(hClipData));
lpszClipData = GlobalLock(hClipData);
lpszMyData = GlobalLock(hMyData);
lstrcpy(lpszMyData, lpszClipData);
GlobalUnlock(hMyData);
GlobalUnlock(hClipData);
```

## **Close the clipboard:**

```
CloseClipboard();
```

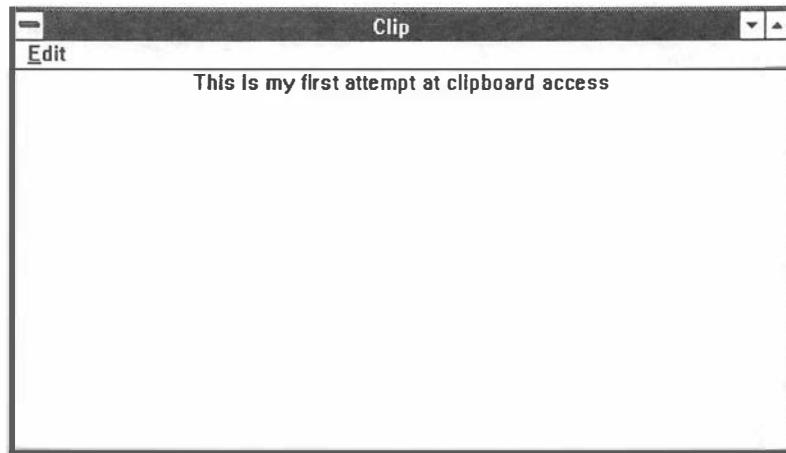
Before text can be passed to the clipboard, the application must make certain that the clipboard can accept data in text format. You can do this by using either the IsClipboardFormatAvailable or GetClipboardData function.

Both functions pass the CF\_TEXT constant. The first function returns TRUE if text exists in the clipboard. This function is one of the few clipboard functions that can be called without opening the clipboard.

Before data can be taken from it, the clipboard must be opened. GetClipboardData provides a handle to the data. This handle equals NULL if the format doesn't match the one currently in the clipboard. This applies only to the calls to GetClipboardData and CloseClipboard because these belong to the clipboard.

Similar to writing to the clipboard, another global memory area must be allocated and the data copied to this new area. First, the area returning the handle of the clipboard must be released and the clipboard must be closed. After this, any other application can be used with the clipboard.

## Example of text format



This example creates a menu that contains the **Cut**, **Copy**, and **Paste** menu items. If you select one of the first two menu items, the text currently in the client area is transferred to the clipboard. The **Paste** command pastes this data into the client window.

If text is already in the clipboard, it is pasted into the client area. This example lets you pass text between different Windows applications (e.g., WRITE.EXE or WINWORD.EXE).

### New function

CloseClipboard  
EmptyClipboard  
GetClipboardData  
GlobalAlloc  
GlobalFree  
GlobalLock  
GlobalReAlloc  
GlobalSize  
GlobalUnlock  
IsClipboardFormatAvailable  
lstrcpy  
OpenClipboard  
SetClipboardData

### Brief description

Close the clipboard  
Clear the clipboard's contents  
Get data from the clipboard  
Get global memory area  
Release global memory area  
Lock global memory area  
Reallocate global memory (new size)  
Compute size of global memory area  
Unlock locked global memory  
TRUE if current format is available  
Copy string to another string  
Open the clipboard  
Set data in clipboard

## Source code: CLIP.C

```
/** CLIP.C ****
** Simple clipboard access
****

#include "windows.h"                                // Include windows.h header file
#include "Clip.h"                                    // Include clip.h header file

BOOL ClipInit ( HANDLE );
long FAR PASCAL ClipWndProc( HWND, unsigned, WORD, LONG);
void WarningBox(void);

/** WinMain (main window function for all Windows applications) ****

int PASCAL WinMain( hInstance, hPrevInstance, lpszCmdLine,cmdShow)
    HANDLE hInstance, hPrevInstance;      // Current & previous instance
    LPSTR  lpszCmdLine;                 // Long ptr after prg. name
    int     cmdShow;                   // App. window's appearance
{
MSG  msg;                                         // Message handle
HWND hWnd;                                       // Window handle

    if (!hPrevInstance)                      // Initialize current instance
    {
        if (!ClipInit( hInstance ))
            return FALSE;
    }

    hWnd = CreateWindow("Clip", "Clip",
                        WS_OVERLAPPEDWINDOW,
                        CW_USEDEFAULT, 0, CW_USEDEFAULT, 0,
                        NULL,
                        NULL,
                        hInstance,
                        NULL);

    ShowWindow( hWnd, cmdShow );           // Make window visible
    UpdateWindow( hWnd );                // Update window

    while (GetMessage(&msg, NULL, 0, 0)) // Message reading
    {
        TranslateMessage(&msg);          // Message translation
        DispatchMessage(&msg);          // Send message to Windows
    }
    return (int)msg.wParam;               // Return wParam of last message
```

```
}

/** Specify window class *****/
BOOL ClipInit( hInstance )
HANDLE hInstance;
{
    WNDCLASS    wcClipClass;

    wcClipClass.hCursor      = LoadCursor( NULL, IDC_ARROW );
    wcClipClass.hIcon        = LoadIcon( NULL, IDI_APPLICATION );
    wcClipClass.lpszMenuName = (LPSTR) "Menu";
    wcClipClass.lpszClassName = (LPSTR) "Clip";
    wcClipClass.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
    wcClipClass.hInstance     = hInstance;
    wcClipClass.style         = CS_VREDRAW | CS_HREDRAW;
    wcClipClass.lpfnWndProc   = ClipWndProc;
    wcClipClass.cbClsExtra    = 0 ;
    wcClipClass.cbWndExtra    = 0 ;

    if (!RegisterClass( &wcClipClass ) )
        return FALSE;
    return TRUE;
}

/** ClipWndProc *****/
/** Main window function: All messages are sent to this window      */
/** *****/
long FAR PASCAL ClipWndProc( hWnd, message, wParam, lParam )
    HWND      hWnd;
    unsigned message;
    WORD      wParam;
    LONG      lParam;
{
    static HMENU    hMenu;
    static HANDLE   hClipData;
    static HANDLE   hMyData;
    char chText[]   =      "This is my first attempt at clipboard access";
    LPSTR    lpszMyData, lpszClipData = NULL;
    static BOOL     bMyData;
    RECT      rect;
    PAINTSTRUCT ps;
    HDC       hdc;

    switch (message)
    {
        case WM_CREATE:           // Create window
```

```
hMenu = GetMenu(hWnd);
if (!(hMyData = GlobalAlloc( GMEM_MOVEABLE
                           |GMEM_NODISCARD, (DWORD)sizeof(chText))))
{
    WarningBox();
    SendMessage(hWnd, WM_CLOSE, 0, 0L);
    return (FALSE);
}
lpszMyData = GlobalLock(hMyData);
lstrcpy(lpszMyData, chText);
GlobalUnlock(hMyData);
bMyData = TRUE;
break;

case WM_INITMENU:           // Initialize menu
    if (IsClipboardFormatAvailable(CF_TEXT))
        EnableMenuItem(wParam, ID_PASTE, MF_ENABLED);
    else
        EnableMenuItem(wParam, ID_PASTE, MF_GRAYED);
    break;

case WM_COMMAND:           // Menu items
    switch(wParam)
    {
        case ID_CUT:
        case ID_COPY:
            if (!(hClipData = GlobalAlloc(GMEM_MOVEABLE
                                         |GMEM_NODISCARD, GlobalSize(hMyData))))
            {
                WarningBox();
                return (FALSE);
            }
            lpszClipData = GlobalLock(hClipData);
            lpszMyData = GlobalLock(hMyData);

            lstrcpy(lpszClipData, lpszMyData);
            GlobalUnlock(hClipData);
            GlobalUnlock(hMyData);

            if (OpenClipboard(hWnd))
            {
                EmptyClipboard();
                SetClipboardData(CF_TEXT,
                                hClipData);
                CloseClipboard();
            }
    }
}
```

```
if (wParam == ID_CUT)
{
    bMyData = FALSE;
    EnableMenuItem(hMenu, ID_CUT,
                   MF_GRAYED);
    EnableMenuItem(hMenu, ID_COPY,
                   MF_GRAYED);
}

InvalidateRect(hWnd, NULL, TRUE);
break;

case ID_PASTE:
    if (OpenClipboard(hWnd))
    {
        if (!(hClipData =
              GetClipboardData(CF_TEXT)))
        {
            CloseClipboard();
            return (FALSE);
        }

        if (!(hMyData =
              GlobalReAlloc( hMyData,
                            GlobalSize(hClipData),
                            GMEM_MOVEABLE)))
        {
            if (!(hMyData =
                  GlobalAlloc(
                    GMEM_MOVEABLE |
                    GMEM_NODISCARD,
                    GlobalSize
                    (hClipData))))
            {
                WarningBox();
                CloseClipboard();
                return (FALSE);
            }
        }
    }

lpszMyData = GlobalLock(hMyData);
lpszClipData = GlobalLock(hClipData);
lstrcpy(lpszMyData, lpszClipData);
GlobalUnlock(hMyData);
GlobalUnlock(hClipData);
CloseClipboard();
EnableMenuItem(hMenu, ID_CUT,
               MF_ENABLED);
```

```
        EnableMenuItem(hMenu, ID_COPY,
                        MF_ENABLED);
        bMyData = TRUE;

        InvalidateRect(hWnd, NULL, TRUE);
    }
    break;

    default:
        return
(DefWindowProc(hWnd,message,wParam,lParam));
    break;
}
break;

case WM_PAINT:           // Client area redraw
    hDC = BeginPaint (hWnd, &ps);

    if (bMyData)
    {
        lpszMyData = GlobalLock (hMyData);
        GetClientRect (hWnd, &rect);
        DrawText (hDC, lpszMyData, -1, &rect,
                  DT_CENTER | DT_WORDBREAK);
        GlobalUnlock (hMyData);
    }
    EndPaint (hWnd, &ps);
    break;

case WM_DESTROY:          // Destroy window
    GlobalFree(hMyData);
    GlobalFree(hClipData);
    PostQuitMessage(0);
    break;

default:
    return (DefWindowProc( hWnd, message, wParam, lParam ));
break;
}

return(0L);
}

void WarningBox(void)
{
    MessageBox( GetFocus(),
                "Not enough memory", "ATTENTION!",
                MB_OK | MB_ICONWARNING);
}
```

```
    MB_ICONHAND | MB_SYSTEMMODAL | MB_OK);  
return;  
}
```

## Module definition file: CLIP.DEF

```
NAME      Clip  
  
DESCRIPTION 'Clipboard demonstration'  
  
EXETYPE   WINDOWS  
  
STUB       'WINSTUB.EXE'  
  
CODE       PRELOAD MOVEABLE  
DATA       PRELOAD MOVEABLE MULTIPLE  
  
HEAPSIZE   4096  
STACKSIZE  4096  
  
EXPORTS    ClipWndProc  @1
```

## Resource script: CLIP.RC

```
#include "clip.h"  
#include "windows.h"  
  
Menu     MENU  
BEGIN  
    POPUP "&Edit"  
        BEGIN  
            MENUITEM "Cu&t",           ID_CUT  
            MENUITEM "&Copy",          ID_COPY  
            MENUITEM "&Paste",          ID_PASTE  
        END  
END
```

## Header file: CLIP.H

```
#define     ID_CUT      11  
#define     ID_COPY     12  
#define     ID_PASTE    13
```

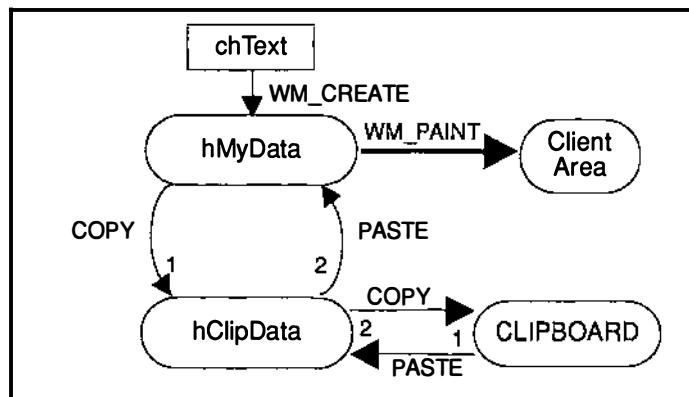
To compile and link this application, use the RCOMPILE.BAT batch file described earlier in this book. Here's the listing again:

```
cl -c -Gw -Zp %1  
rc -r %1.rc  
link /align:16 %1,%1.exe,,libw+slibcew,%1.def  
rc %1.res
```

Save this file to a directory contained in your path or the directory containing your source, module definition and resource scripts. Type the following and press **Enter** to compile CLIP:

```
rcompile clip
```

## How CLIP.EXE works



This application contains two global memory areas. One is used for the text data passed to and from the clipboard, and the other is used for the text that is written to the client area.

Two global areas must be available so that the handle passed from or taken by the clipboard won't belong to an application. This will also allow the clipboard to be used as little as possible.

These same areas can be used for the **Copy** menu item, with an initialization text defined by the chText variable, which is placed in the client area. The flow of data, indicated by the handles, looks similar to the above illustration.

For the initialization text, the WM\_CREATE message provides a global memory block through the GlobalAlloc function. The application displays a message box if no more memory is available.

After this box appears, the application ends by sending the WM\_CLOSE message. Otherwise, the lstrcpy function can disable the lock on the text in this memory area. The lstrcpy function can also work with mixed pointers. This is necessary because chText is a near pointer and lpszMyData is a far pointer.

In Windows, locked global memory blocks should be released as quickly as possible using GlobalUnlock. You'll need a menu handle (generated by GetMenu) to change the status of each menu item. The Boolean bMyData variable contains the value TRUE to indicate that the text was displayed by the WM\_PAINT message.

The WM\_INITMENU message is sent shortly before the menu opens. If the clipboard contains data in text format, the application activates the Paste menu item. This can be determined with the IsClipboardFormatAvailable function and applied to the status of each menu entry.

The Cut and Copy menu items, which are usually found in clipboard-based applications, are identified through the hMyData handle. Selecting either command places the text itself (or a copy of the text) in a new global area, through the hClipData handle.

The size of the new memory area is calculated through the hMyData handle by the GlobalSize function. This handle closes after it successfully opens and empties the clipboard data through the SetClipboardData function. If the user selects the Cut menu item, three things occur:

The Cut and Copy menu items become inactive.

The bMyData Boolean variable changes status.

On the next occurrence of the WM\_PAINT message, the text will no longer appear in the client area.

If the user selects the Paste menu item, the text is pasted from the clipboard. The GetClipboardData function performs this task. The data, taken by its handle, is pasted in for display. Although a memory area is

available, it isn't the proper size. The GlobalReAlloc function sets the new dimensions.

The first time the application calls GlobalReAlloc, a value of NULL is returned because memory wasn't allocated before the call. The GlobalAlloc function receives a new memory area. This copies the text similar to the **Copy** menu item (but in the opposite direction). This must be done before the clipboard can be closed and the handle removed.

Finally, the **Paste** menu item changes the other entries in the MF\_ENABLED routine, and the bMyData Boolean variable is set to TRUE. The next WM\_PAINT message displays the new text, instructed by the InvalidateRect function.

The DrawText function performs this task. Since this function requires a pointer to the text, the global memory area must first be locked with GlobalLock. The entire client area is designated for the DrawText function and the size of the client area is calculated through GetClientRect.

The GlobalFree function releases the global memory areas and the application ends. The handles are no longer valid when this occurs.

## Bitmap format

The CF\_BITMAP identifier allows the clipboard to exchange bitmaps. The procedure is very similar to the text transfer previously described. However, bitmap management is more complex because bitmap structure is more complicated than text structure.

## Writing a bitmap to the clipboard

The process of placing a bitmap in the clipboard is almost identical to placing text in the clipboard. A handle to the bitmap you want to place in the clipboard is needed. Unlike text, a global memory allocation isn't required. The SetClipboardData function requests format information from CF\_BITMAP.

## **Obtaining a bitmap from the clipboard**

To obtain a bitmap from the clipboard, first you must open the clipboard by using the OpenClipboard function.

### **Open the clipboard:**

```
OpenClipboard(hWnd) ;
```

### **Get handle to global memory area from clipboard:**

```
hClipBit = GetClipboardData(CF_BITMAP) ;
```

### **Display bitmap on the screen:**

```
hDC = GetDC(hWnd) ;
hMemDC = CreateCompatibleDC(hDC)
hOldBit = SelectObject(hMemDC, hClipBit);
GetObject(hClipData, sizeof(BITMAP), &bm);
BitBlt(hDC, 20, 20, bm.bmWidth, bm.bmHeight, hMemDC, 0, 0, SRCCOPY) ;
SelectObject(hMemDC, hOldBit);
DeleteDC(hMem, hDC);
ReleaseDC(hWnd, hDC) ;
```

### **Close clipboard:**

```
CloseClipboard();
```

The GetClipboardData function gets the bitmap data, using the bitmap's handle. However, before it can display the bitmap (e.g., using BitBlt), a memory device context must be called. To do this, use the CreateCompatibleDC function.

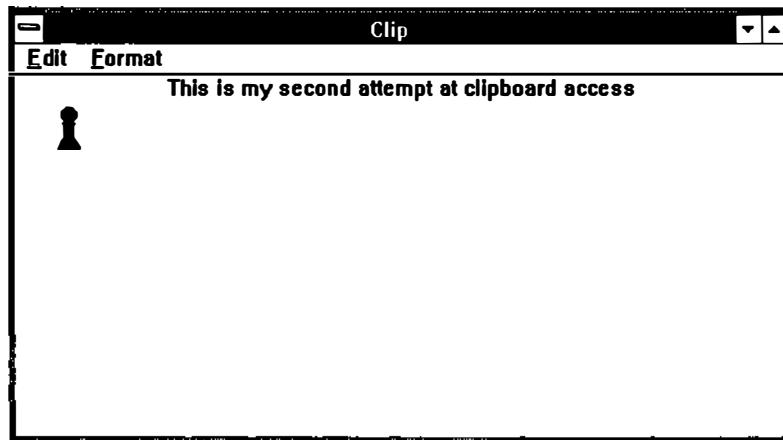
The SelectObject function writes the bitmap handle contained in the clipboard. The GetObject function must calculate the bitmap's dimensions for correct display. This function fills a variable of structure BITMAP with data. The BitBlt output function contains the bitmap's width and height.

The memory device context acts as the source device context, while the display context acts as the target device context. After the bitmap is displayed, the old bitmap returns to the memory device context, and the application releases both device contexts.

`CloseClipboard` must be used to close the bitmap handle. If the `WM_PAINT` message processes the output first message, the handle can be closed in the same way you close a text handle. The application consists of two bitmap handles. One handle represents the clipboard, while the other represents the display.

Bitmaps cannot be copied using the `lstrcpy` function but can be passed using two memory device contexts (see the `CLIPBIT` example below).

## Example of bitmap format



This application is based on the `CLIP.EXE` application listed earlier in this chapter. `CLIPBIT` lets you exchange text and bitmaps between Windows applications. The menu items fit the needs of the format.

You'll also find an initialization bitmap, which appears in the client area when the application starts.

## Source code: CLIPBIT.C

```
/** CLIPBIT.C ****
** Clipboard access for text and bitmap data
*/
****

#include "windows.h"                                // Include windows.h header file
#include "Clipbit.h"                                 // Include clipbit.h header file

BOOL ClipInit ( HANDLE );
long FAR PASCAL ClipWndProc( HWND, unsigned, WORD, LONG);
void WarningBox(void);

/** WinMain ****

int PASCAL WinMain( hInstance, hPrevInstance, lpszCmdLine,cmdShow)
    HANDLE hInstance, hPrevInstance;      // Current & previous instance
    LPTSTR lpszCmdLine;                  // Long ptr to after prg.name
    int cmdShow;                        // App. window's appearance
{
MSG  msg;                                         // Message handle
HWND hWnd;                                        // Window handle

    if (!hPrevInstance)                   // Initialize current instance
    {
        if (!ClipInit( hInstance ))
            return FALSE;
    }

/** Specify appearance of application's main window ****

hWnd = CreateWindow("Clip", "Clip",
    WS_OVERLAPPEDWINDOW,
    CW_USEDEFAULT, 0, CW_USEDEFAULT, 0,
    NULL,
    NULL,
    hInstance,
    NULL);

ShowWindow( hWnd, cmdShow );                      // Make window visible
UpdateWindow( hWnd );                            // Update window

while (GetMessage(&msg, NULL, 0, 0)) // Message reading
{
    TranslateMessage(&msg);                // Message translation
    DispatchMessage(&msg);                // Send message to Windows
```

```
    }
    return (int)msg.wParam;                                // Return wParam of last message
}

BOOL ClipInit( hInstance )                                // Clipboard initialization
HANDLE hInstance;
{
    /** Specify window class *****/
    WNDCLASS    wcClipClass;

    wcClipClass.hCursor      = LoadCursor( NULL, IDC_ARROW );
    wcClipClass.hIcon        = LoadIcon( NULL, IDI_APPLICATION );
    wcClipClass.lpszMenuName = (LPSTR)"Menu";
    wcClipClass.lpszClassName = (LPSTR)"Clip";
    wcClipClass.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
    wcClipClass.hInstance     = hInstance;
    wcClipClass.style         = CS_VREDRAW | CS_HREDRAW;
    wcClipClass.lpfnWndProc   = ClipWndProc;
    wcClipClass.cbClsExtra    = 0 ;
    wcClipClass.cbWndExtra    = 0 ;

    if (!RegisterClass( &wcClipClass ) )
        return FALSE;
    return TRUE;
}

/** ClipWndProc *****/
/** Main window function: All messages are sent to this window */
/** *****/
long FAR PASCAL ClipWndProc( hWnd, message, wParam, lParam )
    HWND      hWnd;
    unsigned message;
    WORD      wParam;
    LONG      lParam;
{
    static HANDLE    hInst;                      // Instance handle
    static HMENU      hMenu;                     // Menu handle
    static HANDLE    hClipData;                  // Clipboard
    static HANDLE    hMyData;                    // handles
    static HBITMAP   hBitmap, hMyBit, hClipBit, hBitOld, hBit2Old;

    char chText[]    = "This is my second attempt at clipboard access";
    LPSTR    lpszMyData, lpszClipData;
    LPSTR    lpszMyBit, lpszClipBit;
    static BOOL      bMyData, bMyBitmap;
```

```
static BOOL      bText = TRUE;

RECT          rect;
PAINTSTRUCT ps;
HDC           hDC, hMemDC, hMem2DC;
BITMAP        bm;

switch (message)
{
    case WM_CREATE:           // Create window
        hInst = GetWindowWord(hWnd, GWW_HINSTANCE);
        hMenu = GetMenu(hWnd);
        if (!(hMyData = GlobalAlloc( GMEM_MOVEABLE
                                      |GMEM_NODISCARD, (DWORD)sizeof(chText))))
        {
            WarningBox();
            SendMessage(hWnd, WM_CLOSE, 0, 0L);
            return (FALSE);
        }
        lpszMyData = GlobalLock(hMyData);
        lstrcpy(lpszMyData, chText);
        GlobalUnlock(hMyData);
                    // Load pawn bitmap
        hBitmap = LoadBitmap(hInst, "Pawn");

        hDC = GetDC(hWnd);
        hMemDC = CreateCompatibleDC( hDC );
        hBitOld = SelectObject(hMemDC,hBitmap);

        hMem2DC = CreateCompatibleDC( hDC );
        hMyBit = CreateCompatibleBitmap(hDC, 32, 32);
        hBit2Old = SelectObject(hMem2DC,hMyBit);

        BitBlt(hMem2DC,0,0,32,32,hMemDC,0,0,SRCCOPY);
        SelectObject(hMemDC,hBitOld);
        SelectObject(hMemDC,hBit2Old);

        DeleteDC(hMemDC);
        DeleteDC(hMem2DC);
        ReleaseDC(hWnd, hDC);

        bMyData = TRUE;
        bMyBitmap = TRUE;

        break;

    case WM_INITMENU:         // Initialize menu
        if (bText)
```

```
{  
    if (IsClipboardFormatAvailable(CF_TEXT))  
        EnableMenuItem(wParam, ID_PASTE,  
                        MF_ENABLED);  
    else  
        EnableMenuItem(wParam, ID_PASTE, MF_GRAYED);  
}  
if (!bText)  
{  
    if (IsClipboardFormatAvailable(CF_BITMAP))  
        EnableMenuItem(wParam, ID_PASTE,  
                        MF_ENABLED);  
    else  
        EnableMenuItem(wParam, ID_PASTE, MF_GRAYED);  
}  
break;  
  
case WM_COMMAND:           // Get messages from menu bar  
    switch(wParam)  
    {  
        case ID_CUT:  
        case ID_COPY:  
            if (bText)  
            {  
                if (!(hClipData =  
                    GlobalAlloc( GMEM_MOVEABLE  
| GMEM_NODISCARD,GlobalSize(hMyData))))  
                {  
                    WarningBox();  
                    return (FALSE);  
                }  
                lpszClipData =  
                    GlobalLock(hClipData);  
                lpszMyData = GlobalLock(hMyData);  
  
                lstrcpy(lpszClipData, lpszMyData);  
                GlobalUnlock(hClipData);  
                GlobalUnlock(hMyData);  
  
                if (OpenClipboard(hWnd))  
                {  
                    EmptyClipboard();  
                    SetClipboardData(CF_TEXT,  
                                    hClipData);  
                    CloseClipboard();  
                }  
            }  
    }  
    if (wParam == ID_CUT)
```

```
{  
    bMyData = FALSE;  
    EnableMenuItem(hMenu,  
        ID_CUT, MF_GRAYED);  
    EnableMenuItem(hMenu,  
        ID_COPY, MF_GRAYED);  
}  
}  
    // for text  
  
if (!bText)  
{  
    hDC = GetDC(hWnd);  
    hMemDC = CreateCompatibleDC( hDC );  
    hBitOld = SelectObject(hMemDC,  
        hMyBit);  
    GetObject(hMyBit,  
        sizeof(BITMAP),  
        (LPSTR)&bm);  
  
    hMem2DC = CreateCompatibleDC( hDC );  
    hClipBit = CreateCompatibleBitmap  
        ( hDC, bm.bmWidth,  
        bm.bmHeight);  
    hBit2Old = SelectObject(hMem2DC,  
        hClipBit);  
  
    BitBlt(hMem2DC, 0, 0, bm.bmWidth, bm.bmHeight, hMemDC, 0, 0, SRCCOPY);  
    SelectObject(hMemDC, hBitOld);  
    SelectObject(hMemDC, hBit2Old);  
  
    DeleteDC(hMemDC);  
    DeleteDC(hMem2DC);  
    ReleaseDC(hWnd, hDC);  
  
    if (OpenClipboard(hWnd))  
    {  
        EmptyClipboard();  
        SetClipboardData(CF_BITMAP, hClipBit);  
        CloseClipboard();  
    }  
    if (wParam == ID_CUT)  
    {  
        bMyBitmap = FALSE;  
        EnableMenuItem(hMenu,  
            ID_CUT, MF_GRAYED);  
        EnableMenuItem(hMenu,  
            ID_COPY, MF_GRAYED);  
    }  
}
```

```
        }

    } // Bitmap

    InvalidateRect(hWnd, NULL, TRUE);
    break;

case ID_PASTE:
    if (bText)
    {
        if (OpenClipboard(hWnd))
        {
            if (!(hClipData =
GetClipboardData(CF_TEXT)))
            {
                CloseClipboard();
                return (FALSE);
            }

            if (!(hMyData =
GlobalReAlloc( hMyData,
GlobalSize(hClipData), GMEM_MOVEABLE)))
            {
                if (!(hMyData =
GlobalAlloc(GMEM_MOVEABLE
| GMEM_NODISCARD, GlobalSize(hClipData))))
                {
                    WarningBox();
                    CloseClipboard();
                    return (FALSE);
                }
            }
        }

        lpszMyData =
            GlobalLock(hMyData);
        lpszClipData =
            GlobalLock(hClipData);
        lstrcpy(lpszMyData,
            lpszClipData);
        GlobalUnlock(hMyData);
        GlobalUnlock(hClipData);
        CloseClipboard();
        EnableMenuItem(hMenu,
            ID_CUT, MF_ENABLED);
        EnableMenuItem(hMenu,
            ID_COPY, MF_ENABLED);
        bMyData = TRUE;
    }
}
```

```
InvalidateRect (hWnd,
                NULL, TRUE);
}
} // Text

if (!bText)
{
    if (OpenClipboard(hWnd))
    {
        if (!(hClipBit =
GetClipboardData(CF_BITMAP)))
        {
            CloseClipboard();
            return (FALSE);
        }

        hDC = GetDC(hWnd);
        hMemDC =
CreateCompatibleDC( hDC );
        hBitOld =
            SelectObject(hMemDC,
            hClipBit);
        GetObject (hClipBit,
            sizeof(BITMAP),
            (LPSTR)&bm);

        hMem2DC =
CreateCompatibleDC( hDC );
        hMyBit =
CreateCompatibleBitmap( hDC, bm.bmWidth,
            bm.bmHeight);
        hBit2Old =
            SelectObject(hMem2DC,
            hMyBit);

        BitBlt (hMem2DC, 0, 0,
            bm.bmWidth,bm.bmHeight,
            hMemDC, 0, 0, SRCCOPY);
        SelectObject (hMemDC,hBitOld);
        SelectObject (hMemDC,hBit2Old);

        DeleteDC(hMemDC);
        DeleteDC(hMem2DC);
        ReleaseDC(hWnd, hDC);

        CloseClipboard();
        EnableMenuItem(hMenu,
            ID_CUT, MF_ENABLED);
    }
}
```

```
        EnableMenuItem(hMenu,
                        ID_COPY, MF_ENABLED);
    }
    bMyBitmap = TRUE;
    InvalidateRect(hWnd, NULL, TRUE);
}

break;

case ID_TEXT:
    bText = TRUE;
    CheckMenuItem(hMenu, ID_TEXT, MF_CHECKED);
    CheckMenuItem(hMenu, ID_BIT, MF_UNCHECKED);
    break;

case ID_BIT:
    bText = FALSE;
    CheckMenuItem(hMenu, ID_TEXT, MF_UNCHECKED);
    CheckMenuItem(hMenu, ID_BIT, MF_CHECKED);
    break;

default:
    return (DefWindowProc(hWnd,message,
                          wParam,lParam));
break;
}
break;

case WM_PAINT:           // Client area redraw
hDC = BeginPaint ( hWnd, &ps);

if (bMyData)
{
    lpszMyData = GlobalLock ( hMyData );
    GetClientRect ( hWnd, &rect );
    DrawText ( hDC, lpszMyData, -1, &rect, DT_CENTER
               | DT_WORDBREAK );
    GlobalUnlock ( hMyData );
}

if (bMyBitmap)
{
    hMemDC = CreateCompatibleDC( hDC );
    hBitOld = SelectObject(hMemDC,hMyBit);
    GetObject(hMyBit, sizeof(BITMAP), (LPSTR)&bm);

BitBlt(hDC,20,20,bm.bmWidth,bm.bmHeight,hMemDC,0,0,SRCCOPY);
    SelectObject (hMemDC,hBitOld);
}
```

```
        DeleteDC(hMemDC);
    }

    EndPaint (hWnd, &ps);
    break;

    case WM_DESTROY:           // Destroy window
        GlobalFree(hMyData);
        GlobalFree(hClipData);
        PostQuitMessage(0);
        break;

    default:
        return (DefWindowProc( hWnd, message, wParam, lParam ));
    break;
}

return(0L);
}

void WarningBox(void)
{
    MessageBox( GetFocus(),
                "Not enough memory", "ATTENTION!",
                MB_ICONHAND | MB_SYSTEMMODAL | MB_OK);
    return;
}
```

## Module definition file: CLIPBIT.DEF

```
NAME      Clipbit

DESCRIPTION 'Clipboard demonstration with bitmap'

EXETYPE   WINDOWS

STUB      'WINSTUB.EXE'

CODE      PRELOAD MOVEABLE
DATA      PRELOAD MOVEABLE MULTIPLE

HEAPSIZE  4096
STACKSIZE 4096

EXPORTS   ClipWndProc  @1
```

## Resource script: CLIPBIT.RC

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

Pawn    BITMAP  pawn.bmp

Menu   MENU
BEGIN
    POPUP  "&Edit"
        BEGIN
            MENUITEM "Cu&t",           ID_CUT
            MENUITEM "&Copy",           ID_COPY
            MENUITEM "&Paste",          ID_PASTE
        END
    POPUP  "&Format"
        BEGIN
            MENUITEM "&Text",          ID_TEXT,      CHECKED
            MENUITEM "&Bitmap",         ID_BIT
        END
END
```

## Header file: CLIPBIT.H

```
#define     ID_CUT      11
#define     ID_COPY     12
#define     ID_PASTE    13
#define     ID_TEXT     21
#define     ID_BIT      22
```

To compile and link this application, use the RCOMPILE.BAT described earlier in this chapter. Here's the listing again:

```
cl -c -Gw -Zp %1
rc -r %1.rc
link /align:16 %1,%1.exe,,libw+slibcew,%1.def
rc %1.res
```

Save this file to a directory contained in your path or the directory containing your source, module definition and resource scripts. Type the following and press **Enter** to compile CLIPBIT:

```
rcompile clipbit
```

## How CLIPBIT.EXE works

The status of the bText Boolean variable determines whether the clipboard handles the data as text or bitmap format. The default is TRUE (i.e., text format). When bText changes, the checkmark moves to the appropriate command in the Format menu.

Processing the WM\_CREATE message loads the 32 pixel by 32 pixel bitmap listed in the CLIPBIT.RC resource script. The bitmap should be displayed using the existing hMyBit handle instead of its own handle. Two memory device contexts copy the bitmap. The bMyBitmap Boolean variable contains TRUE if displaying the bitmap in the client area is allowed.

Before the Edit menu can open, the IsClipboardFormatAvailable function must be called to ensure that the Paste menu item's status is correct. If a bitmap or text exists in the clipboard, this menu item will be enabled. Otherwise, this item will be grayed.

The Cut and Copy menu items use two memory device contexts. The first selects the available bitmap, while the second selects an empty bitmap created by the CreateCompatibleBitmap function. This empty bitmap is accessible through the hClipBit handle.

The GetObject function calculates the size of the bitmap to be passed because the BitBlt function requires the width and height parameters. Since the bitmap handles remain valid, both memory device contexts are closed. The hClipBit handle can now be passed to the clipboard.

When the wParam parameter contains the value ID\_CUT, both menu items assume NF\_GRAYED status. Also, when bMyBitmap receives a value of FALSE, bitmap output cannot be enabled.

When the user selects the Paste menu item, the process is the same as with the Cut and Copy items, except that the copy direction is reversed (i.e., the data is pasted).

The bitmap output occurs with the BitBlt function. In this case, the target display context becomes the display context provided for the BeginPaint function. Only one memory device context is needed to hold the bitmap that will be displayed.

## Additional information about formats

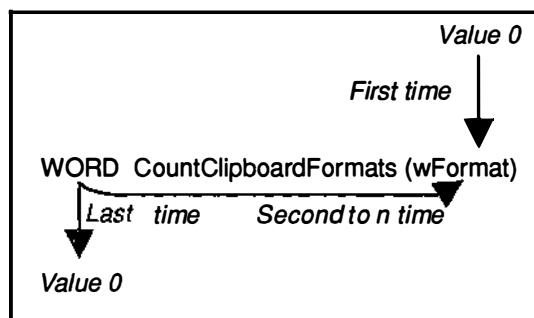
Many applications use the clipboard to exchange text and bitmap data. However, the Windows clipboard isn't very flexible, especially when the user makes special demands on it. Fortunately, the programmer can control almost all facets of clipboard data exchange.

### Multiple formats in the clipboard

As soon as the user opens the clipboard to accept data, the `EmptyClipboard` function empties the clipboard's contents. This prevents the new data from being accidentally appended to existing data. However, it's possible to force the clipboard to store two different formats simultaneously (e.g., text and bitmap). The `SetClipboardData` function can toggle between `EmptyClipboard` and `CloseClipboard`, which enables the user to place a different format in the clipboard.

```
OpenClipboard(hWnd);
EmptyClipboard();
SetClipboardData (CF_TEXT, hClipData);
SetClipboardData (CF_SYLK, hSylk);
SetClipboardData (CF_BITMAP, hBit);
CloseClipboard();
```

The `CountClipboardFormats` function provides the number of different formats in this clipboard when you select the **Paste** menu item. The `EnumClipboardFormats` function helps you determine which format is currently in the clipboard.



To obtain the first format, the value must equal zero. If this function returns a value of zero, then the clipboard contains all available formats. Otherwise, the system defaults to the format that was last returned.

## Delayed rendering

The clipboard is capable of exchanging large amounts of information. This data moves to global memory, which may be in different locations depending on the circumstances. Often, the data may not be taken from an application into the clipboard immediately.

Instead, the data may remain unused in the global memory block for awhile. To avoid this situation, use the delayed rendering technique. This assigns a NULL handle to a global area and passes this handle to SetClipboardData.

When another application calls GetClipboardData to take the data from the clipboard, Windows passes the NULL handle. Windows then sends the WM\_RENDERFORMAT message to the clipboard owner. The clipboard owner is the last window that had the clipboard data.

The message allocates a global memory area for the clipboard owner and copies the data to that area. The wParam parameter states the desired format. The handle is then passed by the SetClipboardData function, without first opening and emptying the clipboard.

In addition to WM\_RENDERFORMAT, there are two other messages for delayed rendering. As soon as another application is called, the EmptyClipboard function assigns a new clipboard owner. The previous owner receives the WM\_DESTROYCLIPBOARD message. So the data doesn't have to be reconstructed for the clipboard.

The WM\_RENDERALLFORMATS message is sent by the current owner application when it ends. The window routine should open and empty the clipboard when it receives this message. The application allocates a global memory area for all the formats that were assigned NULL handles up to this point. The handles received are written to the clipboard by SetClipboardData. The routine ends by closing the clipboard.

## Your own data formats

The clipboard also offers private data formats. The Microsoft Windows Write application has its own data format, which includes font and formatting information. The private data format is used to exchange data between different instances of the same application or to move new format names to different applications.

To work with a private data format, the RegisterClipboardFormat function must specify a freely definable format name. The return value of this call is the new registered format, which can be used by the next clipboard call as a predefined format name.

The hexadecimal value for this new format can range from C000H to FFFFH. If two different applications use the same names for registering a new format, RegisterClipboardFormat supplies the same format code (i.e., both applications can exchange data in this private data format).

The private data format can be displayed in its client area by a clipboard viewer, such as CLPBRD.EXE. The display is enabled, either by one of the predefined display formats, or by the CF\_OWNERDISPLAY format. The resulting messages can then be processed.

The following are the display formats:

CF\_DSPTEXT  
CF\_DSPBITMAP  
CF\_DSPMETAFILEPICT

If data is placed in the clipboard, the private format and one of these display formats are copied to the clipboard. The clipboard viewer selects the display format needed and then interprets the data as text, bitmap or metafile. Finally, it displays the replacement private data format.

However, the clipboard viewer can also take complete control of the data display in the client area. This occurs when formatted text should be displayed, but the formatting didn't transfer to the clipboard. When this happens, the SetClipboardData function selects the CF\_OWNERDISPLAY parameter, and then passes a NULL handle.

Along with the three delayed rendering messages, five additional messages send clipboard viewer and clipboard owner information:

Message	Meaning
WM_ASKCBFORMATNAME	CB viewer sends a message requesting format names.
WM_PAINTCLIPBOARD	CB viewer sends a message stating when new client area should be redrawn.
WM_SIZECLIPBOARD	Size change to CB viewer client area.
WM_VSCROLLCLIPBOARD	Scroll CB viewer window vertically.
WM_HSCROLLCLIPBOARD	Scroll CB viewer window horizontally.

## Clipboard viewer

A clipboard viewer is a Windows application that updates itself when clipboard contents change. The CLIPBRD.EXE application is an example of a clipboard viewer. Messages change the contents. Although several viewers can exist within the Windows system, Windows considers only one viewer—the current clipboard viewer.

Windows takes the window handle from this viewer only, and sends the appropriate clipboard messages to this viewer only. If any other viewers detect changes to the clipboard, a viewer chain opens and documents all existing viewers. So, every viewer listed in the chain receives the messages.

As soon as SetClipboardViewer calls an application, Windows makes this application the current viewer until the function calls another application. SetClipboardViewer provides the handle of the last current viewer. If the handle contains a value of NULL, an earlier viewer doesn't exist in the chain.

This window handle must be stored because the clipboard viewer messages (received either from Windows or from the current viewer) must be re-sent to this window, which is the ancestor in the viewer chain. The SendMessage function is used to send these messages.

The WM\_DRAWCLIPBOARD message is a type of message that is resent when it is the first viewer in a chain. Windows always sends this message to the current clipboard viewer when the clipboard's contents change.

The viewer is mainly responsible for displaying clipboard contents. So, WM\_DRAWCLIPBOARD receives a WM\_PAINT message created by the InvalidateRect function. WM\_PAINT can open the clipboard and GetClipboardData can accept and display the new data.

Later, when a viewer application ends, it must be removed from the viewer chain so that messages won't be sent to the application. The ChangeClipboardChain function does this by giving its own window handle and assigning previous viewers. Basically, this function sends Windows a WM\_CHANGECHAIN message to the current viewer.

The wParam parameter contains the handle to the window about to be removed from the chain and the low word of the lParam parameter contains the ancestor. Therefore, when the ChangeClipboardChain function gives exactly two parameters, these are redirected with the message. Also, WM\_CHANGECHAIN must be passed through the entire viewer chain.

Every viewer must check whether the handle stored in wParam is equal to the stored handle of the ancestor viewer. If it is, the stored handle is made a new viewer and its handle is placed in lParam.

The following table shows the changes that occur when a viewer is removed from the viewer chain. This table assumes that the viewer chain contains six viewers. Viewer 1 is the first viewer, Viewer 2 the second, etc.

Viewer 6 is the current viewer and receives viewer messages directly from Windows. The application represented by Viewer 3 ends and is removed from the chain.

## Old status:

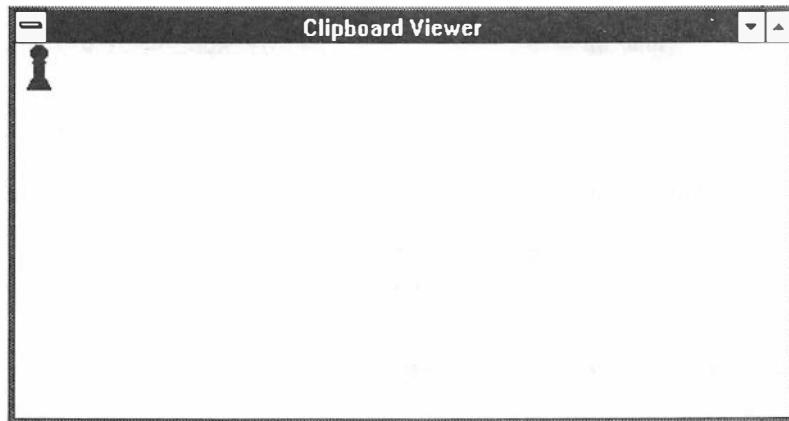
Viewer chain	Stored ancestor handle
Viewer1: hWnd1	NULL
Viewer2: hWnd2	hWnd1
Viewer3: hWnd3	hWnd2
Viewer4: hWnd4	hWnd3
Viewer5: hWnd5	hWnd4
Current viewer: Viewer6: hWnd6	hWnd5

## New status:

Viewer chain	Stored ancestor handle
Viewer1: hWnd1	NULL
Viewer2: hWnd2	hWnd1
Viewer4: hWnd4	hWnd2
Viewer5: hWnd5	hWnd4
Current viewer: Viewer6: hWnd6	hWnd5

After the system sends the WM\_CHANGECHAIN message, Viewer 2 becomes the new ancestor viewer to Viewer 4.

## Example of a clipboard viewer



The following application displays the current clipboard contents (provided the clipboard contains data in text or bitmap format) in its client area.

**New messages**                   **Brief description**

WM\_CHANGECHAIN                   Sends changes to the viewer chain  
WM\_DRAWCLIPBOARD                Sends changes to the clipboard

**New functions**                   **Brief description**

ChangeClipboardChain             Changes the viewer chain  
SetClipboardViewer                Inserts new viewers

## Source code: VIEWER.C

```
/** VIEWER.C ****
/** Displays clipboard contents in either text or bitmap format       */
/****

#include "windows.h"                                                           // Include windows.h header file

long FAR PASCAL ViewerWndProc (HWND, unsigned, WORD, LONG) ;
BOOL ViewerInit (HANDLE);

/** WinMain ****

int PASCAL WinMain (hInstance, hPrevInstance, lpszCmdLine, nCmdShow)
    HANDLE hInstance, hPrevInstance ;                                    // Current & previous instance
    LPSTR lpszCmdLine ;                                                    // Ptr to string after prg. name
    int nCmdShow ;                                                        // App. window's appearance
{
    HWND hWnd ;                                                            // Window handle
    MSG msg ;                                                            // Message handle

    if (!hPrevInstance)                                                    // Initialize current instance
    {
        if (!ViewerInit( hInstance ))
            return FALSE;
    }

/** Specify appearance of application's main window ****

hWnd = CreateWindow ("Viewer",
                     "Clipboard Viewer",
                     WS_OVERLAPPEDWINDOW,
```

```
CW_USEDEFAULT, 0,
CW_USEDEFAULT, 0,
NULL,
NULL,
hInstance,
NULL);

ShowWindow (hWnd, nCmdShow);           // Make window visible
UpdateWindow (hWnd);                 // Update window

while (GetMessage (&msg, NULL, 0, 0)) // Message reading
{
    TranslateMessage (&msg);         // Message translation
    DispatchMessage (&msg);         // Send message to Windows
}
return msg.wParam;                   // Return wParam of last message
}

BOOL ViewerInit( hInstance )          // Viewer initialization
HANDLE hInstance;                   // Instance handle
{

/** Specify window class *****/
WNDCLASS   wcViewerClass;

wcViewerClass.hCursor      = LoadCursor( NULL, IDC_ARROW );
wcViewerClass.hIcon        = LoadIcon( NULL, IDI_APPLICATION );
wcViewerClass.lpszMenuName = NULL;
wcViewerClass.lpszClassName = (LPSTR)"Viewer";
wcViewerClass.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
wcViewerClass.hInstance     = hInstance;
wcViewerClass.style        = CS_VREDRAW | CS_HREDRAW;
wcViewerClass.lpfnWndProc  = ViewerWndProc;
wcViewerClass.cbClsExtra   = 0;
wcViewerClass.cbWndExtra   = 0;

if (!RegisterClass( &wcViewerClass ) )
    return FALSE;
return TRUE;
}

/** ViewerWndProc *****/
/** Main window function: All messages are sent to this window      */
*****
```

```
long FAR PASCAL ViewerWndProc (hWnd, message, wParam, lParam)
    HWND   hWnd;
```

```
unsigned message;
WORD      wParam;
LONG      lParam;
{

    static HWND      hWndPrevViewer;      // Handle for previous viewer
    HANDLE   hText, hBitmap, hBitOld;
    HDC      hDC, hMemDC;
    LPSTR   lpszText;
    BITMAP  bm;
    PAINTSTRUCT ps;
    RECT    rect;

    switch (message)
    {
        case WM_CREATE:                  // Create window
            hWndPrevViewer = SetClipboardViewer (hWnd) ;
            break ;

        case WM_CHANGECBCHAIN :         // Change clipboard chain
            if (wParam == hWndPrevViewer)
                hWndPrevViewer = LOWORD (lParam) ;
            else
                SendMessage (hWndPrevViewer, message, wParam,
lParam) ;
            break ;

        case WM_DRAWCLIPBOARD :
            if (hWndPrevViewer)
                SendMessage (hWndPrevViewer, message, wParam,
lParam);
            InvalidateRect (hWnd, NULL, TRUE) ;
            break;

        case WM_PAINT:                  // Client area redraw
            hDC = BeginPaint (hWnd, &ps) ;
            GetClientRect (hWnd, &rect) ;
            OpenClipboard (hWnd) ;

            if (hText = GetClipboardData (CF_TEXT))
            {
                lpszText = GlobalLock (hText) ;
                DrawText (hDC, lpszText, -1, &rect, DT_WORDBREAK) ;
                GlobalUnlock (hText) ;
            }
            if (hBitmap = GetClipboardData (CF_BITMAP))
            {

```

```
    hMemDC = CreateCompatibleDC( hDC );
    hBitOld = SelectObject(hMemDC,hBitmap);
    GetObject(hBitmap, sizeof(BITMAP), (LPSTR)&bm);

BitBlt(hDC,0,0,bm.bmWidth,bm.bmHeight,hMemDC,0,0,SRCCOPY);
    SelectObject (hMemDC,hBitOld);
    DeleteDC(hMemDC);
}

CloseClipboard () ;
EndPaint (hWnd, &ps) ;
break ;

case WM_DESTROY:           // Destroy window
    ChangeClipboardChain (hWnd, hWndPrevViewer) ;
    PostQuitMessage (0) ;
break ;

default:
    return DefWindowProc (hWnd, message, wParam, lParam);
}
return 0L ;
}
```

## Module definition file: VIEWER.DEF

NAME	VIEWER
DESCRIPTION	'Clipboard Viewer Example'
EXETYPE	WINDOWS
STUB	'WINSTUB.EXE'
CODE	PRELOAD MOVEABLE
DATA	PRELOAD MOVEABLE MULTIPLE
HEAPSIZE	4096
STACKSIZE	4096
EXPORTS	ViewerWndProc

To compile and link this application, use the COMPILE.BAT described earlier in this chapter. Here's the listing again:

```
cl -c -Gw -Zp %1  
link /align:16 %1,%1.exe,,libw+slibcew,%1.def  
rc %1.exe
```

Save this file to a directory contained in your path or the directory containing your source, module definition and resource scripts. Type the following and press **Enter** to compile VIEWER:

```
compile viewer
```

## How VIEWER.EXE works

When the WM\_CREATE message occurs, the SetClipboardViewer function adds the window to the viewer chain. As soon as the clipboard's contents change, the window routine receives the WM\_DRAWCLIPBOARD message, from which the ancestor viewer can be determined.

A WM\_PAINT message generates space for displaying the contents with InvalidateRect. After the WM\_PAINT message opens the clipboard, the application checks for text or bitmap format. If either case is true, the data is prepared and displayed.

If any other viewers are removed from the chain, the WM\_CHANGECHAIN changes viewer status. When the application ends, it calls the ChangeClipboardChain function.

# File Management

## Overview

Many Windows applications can generate different types of data. This data is placed in files for later recall or display. Some files initialize applications (i.e., pass starting values to these applications). If these values change because of upgrading, a new initialization file is needed instead of a software upgrade.

In the following sections we'll discuss MS-DOS files, as well as initialization files.

## MS-DOS files

Windows is capable of multitasking (i.e., running several applications or several instances of one application). However, certain rules must be followed.

First, a file shouldn't be open when an application calls the GetMessage or PeekMessage functions. In certain instances, if this isn't done files may be deleted. Otherwise, you can easily exceed the open file limits set by MS-DOS. So, any file can be opened or closed by using the same message processing.

The file must also follow DOS file handling conventions. The filename can contain up to eight characters and have an optional three character file extension. Instead of the ANSI (Windows) character set, the filename must be entered using the OEM (system resident) character set.

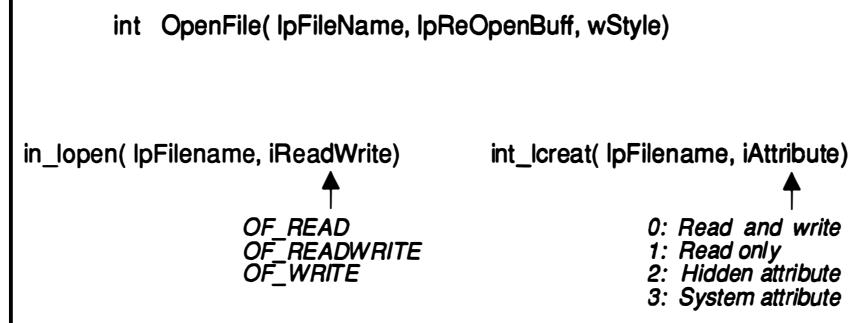
If a file uses temporary files, each instance must have its own file. The GetTempFilename function can be used to assign clearly distinguishable filenames.

When task switching, if a message box cannot be displayed in another application while files are open, every message box should be treated as system modal boxes.

File access must be performed through Windows functions rather than normal C functions. The Windows functions are model independent.

## OpenFile function

Files can be opened with either the OpenFile function or the \_lopen function. OpenFile offers more options (e.g., for opening a new file). The \_lcreat function can also open new files.



The first parameter of the OpenFile function is a pointer to the filename. If the file doesn't exist in the current directory, the PATH function instructs OpenFile to search the path.

Filenames are converted from ANSI characters to strings based on the OEM character set. If you prefer to use the \_lopen and \_lcreat functions, you must specify this conversion with the AnsiToOem function.

The second parameter of the OpenFile function is a pointer to a variable of structure OFSTRUCT. The elements of this structure are filled from the first function call. They can then be used for all subsequent OpenFile calls to the same file, as long as the OF\_REOPEN flag is set.

The OFSTRUCT looks as follows:

## OFSTRUCT:

Array	Data type	Description
cBytes	BYTE	Byte length of structure
fFixedDisk	BYTE	I0 for files on hard drive
nErrCode	WORD	MS-DOS error code
reserved[4]	BYTE	Reserved
szPathName[120]	BYTE	Complete filename

Once OpenFile is executed successfully, szPathName is filled with the entire filename, including the drive and path.

The last parameter in OpenFile indicates the type of access to be performed on the file. Logical OR operators can be used to combine accesses:

OF_CREATE	Create new file
OF_DELETE	Delete file
OF_REOPEN	Open file with OFSTRUCT structure
OF_READ	Open file for reading only
OF_READWRITE	Open file for reading and writing
OF_WRITE	Open file for writing only
OF_EXIST	Check for existing file
OF_PARSE	Parse OFSTRUCT structure
OF_VERIFY	Compare current file's date with earlier file
OF_PROMPT	Display message box if file doesn't exist
OF_CANCEL	Add Cancel button to message box

When the OF\_CREATE value is set, the application attempts to create a new file with the specified name. If a file with this name already exists, the size of this file will be set to zero.

You can avoid this data loss by including the OF\_EXIST flag. This flag will check for the existence of a file with the same name. The OpenFile function then returns a value of -1.

Opening an existing file requires additional specifications. The file can be opened for reading only, for writing only, or for reading and writing. The value returned to the OpenFile function is the handle to the open file. All subsequent file calls use this handle for access. If the file couldn't be opened, the return value is -1.

The OF\_PROMPT value generates a dialog box that informs the user that the file doesn't exist, and prompts the user to insert a diskette in drive A:. The OF\_CANCEL value places a Cancel button in the OF\_PROMPT dialog box. If the file doesn't exist, the user then has the option of cancelling the procedure.

Frequently the file can be reopened by using the OF\_REOPEN value. The filename isn't necessary because the complete name and path were taken from the OFSTRUCT structure the first time the file was opened. This enables the application to find the file when the user has changed the current directory, and when the file you want isn't in the DOS PATH variable.

The OpenFile value can delete a file of the specified name when the OF\_DELETE function is called. This value returns -1 if the deletion couldn't be executed.

## Closing a file

As soon as you're finished processing a file, it should be closed immediately using the \_lclose function.

```
int _lclose( hFile)
```

The \_lclose function requires only one parameter - the file handle. If the file closes successfully, \_lclose returns a value of 0. If an error occurs, -1 is returned.

## Reading a file

Data can be read from an open file if you've set the read attribute. Windows uses the following function for this task:

```
int _lread( hFile, lpBuffer, wBytes)
```

The hFile parameter indicates which file should be read. A buffer receives the file data for further processing.

Since files are often large, they aren't read into memory as a single unit. The lpBuffer parameter indicates the size of the buffer allocated for this

file data. The wBytes parameter determines the number of bytes loaded in at a time.

The \_lread function returns the number of bytes read. Remember that this number may not always match the number stored in the wBytes parameter. The value returned will probably be smaller than the wBytes parameter if the end of file has been reached. If an error occurs, the \_lread function returns -1.

## Writing to a file

The parameters used to read a file are also used to write to a file. Windows uses the following function to write to a file:

```
int _lwrite( hFile, lpBuffer, wBytes)
```

The hFile parameter indicates the file handle. The data itself is in a buffer, whose size is specified by the lpBuffer parameter. The wBytes parameter determines the number of bytes to be transferred.

The \_lwrite function returns the number of bytes that are currently written to the file.

## Setting the file pointer

The file pointer automatically adjusts to its new current position in the file after every read and write operation.

When you first open a file, the file pointer points to the start of the file. The file pointer's position must be changed if you wish to add data to another part of the file (e.g., if you append data to the end of the file).

Windows uses the \_lseek function for this purpose.

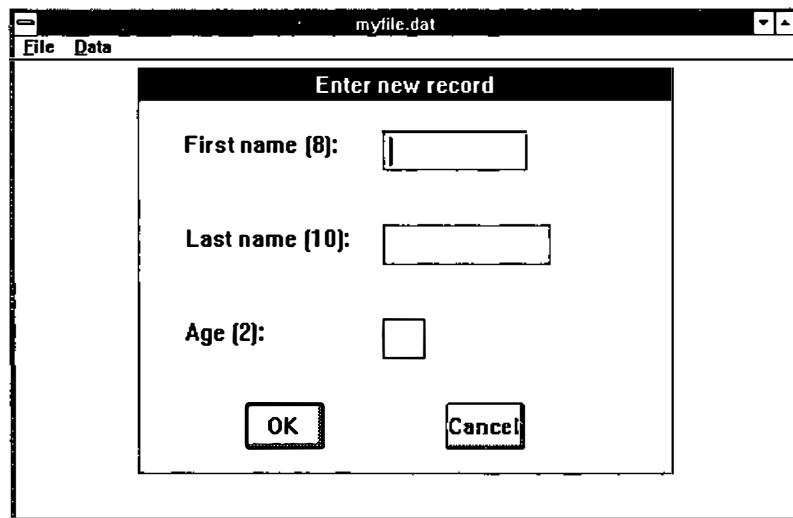
```
LONG _lseek( hFile, lOffset, iOrigin)
```

- 0 Startposition = file beginning
- 1 Startposition = current position
- 2 Startposition = End of file

Where the pointer is located depends on three starting positions: Start of file, end of file, and current position. The second parameter contains the desired start position. The third parameter contains the number of bytes the pointer should move. If the pointer needs positioning from the end of file, the pointer moves toward the start of file.

The \_lseek function returns the file pointer's new position in bytes, counting from the beginning of the file.

## Example of file access



This application demonstrates simple file operations. It consists of a menu from which the user can create a new file, open an existing file, enter data, and delete a file. When the user selects the New or Open menu items, a dialog box containing an edit control appears. The user can enter a filename without a path specification in this dialog box.

Once a file is open, selecting the Enter menu item allows the user to enter text records in this file. A dialog box appears, providing edit controls for first names, last names, and ages. Clicking on the  button saves this data to the file and displays it in the main window.

New message

Brief description

EM\_LIMITTEXT

Sets maximum text size for edit control

**New functions**

	<b>Brief description</b>
_lclose	Closes a file
_lseek	Sets the file pointer
lread	Reads data from a file
lwrite	Writes data to a file
OpenFile	Opens, creates, and deletes a file

**New structures**

	<b>Brief description</b>
OFSTRUCT	OpenFile structure

**Source code: FILE.C**

```
/** FILE.C ****
** Simple database, storing names and ages. This application uses dialog **
** boxes and message boxes.                                              */
****

#include "windows.h"
#include "file.h"
#include "string.h"

BOOL FileInit ( HANDLE );
long FAR PASCAL FileWndProc( HWND, unsigned, WORD, LONG);
BOOL FAR PASCAL OpenNewDlgProc( HWND, unsigned, WORD, LONG);           // Function - Open/New dialog box
BOOL FAR PASCAL EntryDlgProc( HWND, unsigned, WORD, LONG);             // Function - Entries dialog box
static char      szNameText[15], szDataRecord[24];

int PASCAL WinMain( hInstance, hPrevInstance, lpszCmdLine,cmdShow)
HANDLE      hInstance, hPrevInstance;
LPSTR       lpszCmdLine;
int         cmdShow;
{
    MSG   msg;
    HWND  hWnd;

    if (!hPrevInstance)
    {
        if (!FileInit( hInstance ))
            return FALSE;

    /** Specify main application window ****
}

```

## File Management

---

```
hWnd = CreateWindow("File",
                     "File management demo",
                     WS_OVERLAPPEDWINDOW,
                     CW_USEDEFAULT, 0,
                     CW_USEDEFAULT, 0,
                     NULL, NULL, hInstance, NULL);

ShowWindow( hWnd, cmdShow );
UpdateWindow( hWnd );

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

BOOL FileInit( hInstance )
HANDLE hInstance;

/** Specify window class *****/
{

    WNDCLASS      wcFileClass;

    wcFileClass.hCursor      = LoadCursor( NULL, IDC_ARROW );
    wcFileClass.hIcon        = LoadIcon( NULL, IDI_APPLICATION );
    wcFileClass.lpszMenuName = (LPSTR)"MenuFile";
    wcFileClass.lpszClassName = (LPSTR)"File";
    wcFileClass.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
    wcFileClass.hInstance     = hInstance;
    wcFileClass.style        = CS_VREDRAW | CS_HREDRAW;
    wcFileClass.lpfnWndProc  = FileWndProc;
    wcFileClass.cbClsExtra   = 0 ;
    wcFileClass.cbWndExtra   = 0 ;

    if (!RegisterClass( &wcFileClass ) )
        return FALSE;
    return TRUE;
}

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

```
static HANDLE hInst;
char lpBuffer[400], lpBuff[1];
FARPROC lpprocDia;
static OFSTRUCT OfStruct;
int hFile, iNumber, i;
BOOL bExist = FALSE;
static BOOL bDFile;
HDC hdc;
PAINTSTRUCT ps;
RECT rect;

switch (message)
{
    case WM_CREATE:
        hInst = GetWindowWord(hWnd, GWW_HINSTANCE);
        bDFile = FALSE;
        memset(szNameText, ' ', 15);
        break;

    case WM_COMMAND: // Menu items
        switch(wParam)
        {
            case ID_DEL: // Delete file & data
                GetWindowText(hWnd, szNameText, 15);
                OpenFile(szNameText, &OfStruct, OF_DELETE);
                SetWindowText(hWnd, "File management demo");
                bDFile = FALSE;
                memset(szNameText, ' ', 15);
                InvalidateRect(hWnd, NULL, TRUE);
                break;

            case ID_NEW: // Create new file
                lpprocDia = MakeProcInstance
                            (OpenNewDlgProc, hInst);
                if (DialogBox(hInst, "OPENNEW", hWnd,
                              lpprocDia))
                {
                    if (OpenFile(szNameText, &OfStruct,
                                 OF_EXIST) >= 0)
                    {
                        bExist = TRUE;
                        if (MessageBox(hWnd,
                                      "Replace existing file?",
                                      "File", MB_OKCANCEL |
                                      MB_ICONHAND) == IDOK)
                        {
                            bExist = FALSE;
                        }
                    }
                }
        }
}
```

```
        }

        if (bExist)
            memset(szNameText, ' ', 15);
        if (!bExist)
        {
            hFile = OpenFile(szNameText,
                             &OfStruct,
                             OF_READWRITE |
                             OF_CREATE);
            if (hFile >= 0)
            {
                lpBuff[0] = '\0';
                _lwrite(hFile,
                        lpBuff, 1);
                _lclose(hFile);
                SetWindowText(hWnd,
                              szNameText);
                bDFile = TRUE;
            }
            else
                MessageBox(hWnd,
                           "Illegal filename", "Attention",
                           MB_OK);
        }
        InvalidateRect(hWnd, NULL, TRUE);
        FreeProcInstance (lpprocDia);
        break;

    case ID_OPEN:// Open existing file
        lpprocDia = MakeProcInstance
                    (OpenNewDlgProc, hInst);
        if (DialogBox(hInst, "OPENNEW", hWnd,
                      lpprocDia))
        {
            if (OpenFile(szNameText,
                         &OfStruct, OF_EXIST) == -1)
            {
                MessageBox(hWnd, "File not
                           found", "Error", MB_OK);
                bDFile = FALSE;
                memset(szNameText, ' ', 15);
            }
            else
            {
```

```
bDFile = TRUE;
SetWindowText(hWnd,
              szNameText);
InvalidateRect(hWnd, NULL,
                  TRUE);
}
}
FreeProcInstance (lpprocDia);
break;

case ID_ENT: // Enter record for placement in file
    lpprocDia = MakeProcInstance (EntryDlgProc,
                                  hInst);
    if (DialogBox(hInst, "ENTRIES", hWnd,
                  lpprocDia))
    {
        GetWindowText(hWnd, szNameText, 15);
        hFile = OpenFile(NULL, &OfStruct,
                          OF_READWRITE |
                          OF_REOPEN);
        if (hFile >= 0)
        {
            i=0;
            lpBuff[0]= 'a';
            while (lpBuff[0] != '\0')
            {
                _llseek(hFile,i,0);
                i++;
                _lread(hFile,
                       lpBuff, 1);
            }
            i--;
            _llseek(hFile,i,0);
            _lwrite(hFile,
                     szDataRecord, 24);
            _lclose(hFile);
            InvalidateRect(hWnd,
                           NULL, TRUE);
        }
        if (hFile == -1)
            MessageBox(hWnd,
                      "No file open", "Error", MB_OK);
    }

    FreeProcInstance (lpprocDia);
    break;
```

```
        }

        break;

    case WM_PAINT:
        hDC = BeginPaint(hWnd, &ps);

        if (bDFile)
        {
            GetWindowText(hWnd, szNameText, 15);

            if (OpenFile(szNameText, &OfStruct, OF_EXIST) == -1)
            {
                bDFile = FALSE;
                SetWindowText(hWnd, "File management demo");
                memset(szNameText, ' ', 15);
            }

            hFile = OpenFile(szNameText, &OfStruct, OF_READ |
                            OF_PROMPT | OF_CANCEL);
            if (hFile >= 0)
            {
                memset(lpBuffer, ' ', 400);
                iNumber = _lread( hFile, lpBuffer, 400);
                if (iNumber >= 0)
                {
                    GetClientRect(hWnd, &rect);
                    DrawText(hDC, lpBuffer,
                            -1, &rect, DT_CENTER);
                }
                _lclose(hFile);
            }
        }
        EndPaint(hWnd, &ps);
        break;

    case WM_DESTROY:
        PostQuitMessage(0);
        break;

    default:
        return (DefWindowProc( hWnd, message, wParam, lParam ));
        break;
    }
    return(0L);
}

/** Generate OPEN/NEW dialog box ****
*****
```

```
BOOL FAR PASCAL OpenNewDlgProc( hDlg, message, wParam, lParam )
HWND      hDlg;
unsigned   message;
WORD      wParam;
LONG      lParam;
{
    static HWND      hName;

    switch (message)
    {
        case WM_INITDIALOG:
            hName = GetDlgItem(hDlg, ID_NAME);
            SendMessage(hName, EM_LIMITTEXT, 12, 0L);
            SetFocus(hName);
            return(FALSE);
            break;

        case WM_COMMAND:
            switch (wParam)
            {
                case IDOK: // OK, open file
                    GetDlgItemText( hDlg, ID_NAME,
                                    szNameText, 15 );
                    EndDialog (hDlg, 1);
                    return(TRUE);
                    break;

                case IDCANCEL: // Cancel, don't open file
                    EndDialog (hDlg, 0);
                    return(TRUE);
                    break;

                default:
                    return(FALSE);
            }
        default:
            return(FALSE);
    }

}

/** Generate Enter data dialog box ****
 */

BOOL FAR PASCAL EntryDlgProc( hDlg, message, wParam, lParam )
HWND      hDlg;
unsigned   message;
WORD      wParam;
LONG      lParam;
{
```

```
static char szName1Text[9], szName2Text[11], szName3Text[3];
HWND           hFirstN, hLastN, hAge;
OFSSTRUCT      OfStruct;
int            hFile;

switch (message)
{
    case WM_INITDIALOG:          // Dialog controls
        hFirstN = GetDlgItem(hDlg, ID_FIRSTNAME);
        hLastN = GetDlgItem(hDlg, ID_LASTNAME);
        hAge = GetDlgItem(hDlg, ID_YOURAGE);
        SendMessage(hFirstN, EM_LIMITTEXT, 8, 0L);
        SendMessage(hLastN, EM_LIMITTEXT, 10, 0L);
        SendMessage(hAge, EM_LIMITTEXT, 2, 0L);
        SetFocus(hFirstN);
        return(FALSE);
        break;

    case WM_COMMAND:             // Buttons
        switch (wParam)
        {
            case IDOK:   // OK, store record
                memset(szName1Text, ' ', 9);
                memset(szName2Text, ' ', 11);
                memset(szName3Text, ' ', 3);

                GetDlgItemText( hDlg,
                                ID_FIRSTNAME,szName1Text, 9);
                GetDlgItemText( hDlg,
                                ID_LASTNAME,szName2Text, 11);
                GetDlgItemText( hDlg,
                                ID_YOURAGE,szName3Text, 3);
                memset(szDataRecord, ' ',24);
                memcpy(szDataRecord, szName1Text,
                       strlen(szName1Text));
                memcpy(szDataRecord+9, szName2Text,
                       strlen(szName2Text));
                memcpy(szDataRecord+20, szName3Text,
                       strlen(szName3Text));
                memcpy(szDataRecord+22, "\n", 2);
                EndDialog (hDlg, 1);
                return(TRUE);
                break;
        }
}
```

```

        case IDCANCEL: // Cancel, don't store record
            EndDialog (hDlg, 0);
            return(TRUE);
            break;

        default:
            return(FALSE);
    }
default:
    return(FALSE);
}

```

## Module definition file: FILE.DEF

```

NAME          File

DESCRIPTION   'File example'

EXETYPE      WINDOWS

STUB          'WINSTUB.EXE'

CODE          PRELOAD MOVEABLE
DATA          PRELOAD MOVEABLE MULTIPLE

HEAPSIZE      4096
STACKSIZE     4096

EXPORTS       FileWndProc    @4
              OpenNewDlgProc  @8
              EntryDlgProc   @9

```

## Resource script: FILE.RC

```

#include "file.h"
#include "windows.h"

#include "opennew.dlg"
#include "entries.dlg"

MenuFile      MENU
BEGIN
    POPUP  "&File"
        BEGIN

```

```
        MENUITEM "&New",           ID_NEW
        MENUITEM "&Open",          ID_OPEN
        MENUITEM "&Delete",         ID_DEL
    END

POPUP "&Data"
BEGIN
    MENUITEM "&Entry",      ID_ENT
END

END
```

## **Header file: FILE.H**

```
#define ID_NEW 10
#define ID_OPEN 11
#define ID_DEL 12
#define ID_ENT 13

#define ID_NAME 20

#define ID_FIRSTNAME 30

#define ID_LASTNAME 31

#define ID_YOURAGE 32
```

## **Dialog box file: ENTRIES.DLG**

```
ENTRIES DIALOG LOADONCALL MOVEABLE DISCARDABLE 52, 49, 162, 111
CAPTION "Enter new record"
STYLE WS_BORDER | WS_CAPTION | WS_DLGFREAME | WS_POPUP
BEGIN
CONTROL "First name (8):", -1, "static", SS_LEFT | WS_CHILD, 14, 9, 49, 8
CONTROL "Last name (10):", -1, "static", SS_LEFT | WS_CHILD, 14, 37, 58, 8
CONTROL "Age (2):", -1, "static", SS_LEFT | WS_CHILD, 14, 65, 45, 8
CONTROL "", ID_FIRSTNAME, "edit", ES_LEFT | ES_AUTOHSCROLL | WS_BORDER | WS_TABSTOP
| WS_CHILD, 74, 9, 44, 12
CONTROL "", ID_LASTNAME, "edit", ES_LEFT | ES_AUTOHSCROLL |
                WS_BORDER | WS_TABSTOP | WS_CHILD, 74, 37, 51, 12
CONTROL "", ID_YOURAGE, "edit", ES_LEFT | ES_AUTOHSCROLL | WS_BORDER |
                WS_TABSTOP | WS_CHILD, 74, 65, 13, 12
CONTROL "OK", IDOK, "button", BS_DEFPUSHBUTTON | WS_TABSTOP |
                WS_CHILD, 32, 90, 24, 14
```

---

```
CONTROL "Cancel", IDCANCEL, "button", BS_PUSHBUTTON | WS_TABSTOP |
    WS_CHILD, 93, 90, 24, 14
END
```

## Dialog box file: OPENNEW.DLG

```
OPENNEW DIALOG LOADONCALL MOVEABLE DISCARDABLE 53, 68, 131, 78
CAPTION "Open / New"
STYLE WS_BORDER | WS_CAPTION | WS_DLGFRADE | WS_POPUP
BEGIN
    CONTROL "Filename:", -1, "static", SS_LEFT | WS_GROUP | WS_CHILD, 31, 15, 49, 8
    CONTROL "", ID_NAME, "edit", ES_LEFT | ES_AUTOHSCROLL | WS_BORDER | WS_TABSTOP |
        WS_CHILD, 30, 25, 61, 12
    CONTROL "OK", IDOK, "button", BS_DEFPUSHBUTTON | WS_GROUP | WS_TABSTOP |
        WS_CHILD, 18, 53, 24, 14
    CONTROL "Cancel", IDCANCEL, "button", BS_PUSHBUTTON | WS_CHILD, 69, 53, 37, 14
END
```

To compile and link this application, use the RCOMPILE.BAT described earlier in this chapter. Here's the listing again:

```
cl -c -Gw -Zp %1
rc -r %1.rc
link /align:16 %1,%1.exe,,libw+slibcew,%1.def
rc %1.res
```

Save this file to a directory contained in your path or the directory containing your source, module definition and resource scripts. Type the following and press **Enter** to compile FILE:

```
rcompile file
```

## How FILE.EXE works

The OPENNEW and ENTRIES dialog boxes enable the user to interact with the application through the edit controls. These controls are assigned ID values that are defined by the FILE.H header file. In both the OpenNewDlgProc and EntryDlgProc dialog box routines, the EM\_LIMITTEXT message limits the amount of text that can be entered in each edit control.

The WINDOWS.H header file defines the IDOK and IDCANCEL dialog buttons. When the user clicks on the **OK** button, the GetDlgItemText function reads the specified file into the corresponding buffer.

Entering data in the First name, Last name, and Age (2) fields stores this information as a record in the szDataRecord string. Every record ends with a carriage return (CR) and "\0". Both dialog box files are added to the FILE.RC resource script by the #include statement.

The WM\_CREATE message passes the current instance's handle with the GetWindowWord function. Several functions need this handle information. The bDFFile Boolean variable contains the value FALSE, which indicates that a file isn't currently open (the WM\_PAINT message needs this information). The szNameText character string contains the filename entered in the OpenNewDlgProc dialog box routine. Initially this string is filled with spaces.

The user can select from four menu items. Selecting New opens a new file. This menu item calls the OPENNEW dialog box, in which the user can enter the new name. When the user selects the **OK** button, the DialogBox function (through EndDialog(hDlg, 1)) receives a value of 1, which is the same as the return value. The OF\_EXIST parameter of the OpenFile function checks for a file with the same name. A value other than -1 occurs if the file already exists.

A message box can query whether the existing file should be deleted. If the user wants to replace the file or enter a new filename, the OpenFile function creates this file. A "\0" is inserted for the DrawText function called by the WM\_PAINT message. The SetWindowText function changes the window title to the current filename. If the OpenFile function contains the return value -1, the filename that was entered may be longer than the application normally allows.

The application can also open an existing file for input. Selecting the Open menu item displays the same dialog box that appears when the user selects New. If the filename entered by the user already exists, the name appears in the window title. The bDFFile Boolean variable is set to TRUE and a WM\_PAINT message generates a window displaying the file's contents.

Selecting the **Delete** menu item deletes the current file. The GetWindowText function takes the name from the window title. The OpenFile function and the OF\_DELETE parameter perform the deletion. The WM\_PAINT message no longer displays the file's contents.

Immediately after creating or opening a file, the user can select **Data Enter** to enter records in the file. If the file must be reopened, the OpenFile function's OF\_REOPEN value handles this. A filename isn't necessary; passing the address of the ofstruct variable before selecting **New** or **Open** is usually sufficient.

The \_lseek function always increments the file pointer by one byte, until the pointer reaches the "\0" (end of file) character. This character is the starting point for the next record, for writing with the \_lwrite function. When the file pointer increments for the \_lread function, the pointer set for writing decrements by one byte. The file then closes.

If the user forgets to open a file before entering data, a message box appears displaying the message "No file open". If the bDFile Boolean variable is set to FALSE, the WM\_PAINT message processing will only delete the old window contents without displaying anything new. This can occur if the user has just selected the **Delete** menu item to delete the file. Otherwise, the message checks whether the file still exists. You can also delete the file using a different Windows application or another instance of the same application.

A negative result removes the filename from the title bar and sets the bDFile variable to FALSE. Any other result reopens the specified file. The 400 byte lpBuffer buffer is filled with spaces and then the file data loads into this buffer. The DrawText function displays the records loaded into the buffer. Every record ends with a CR. The output ends when the application receives a "\0" from the file.

## Initialization Files

Initialization files have the .INI extension. These files contain parameter information for other files. For example, an initialization file can configure Windows to different environments for different users. These values can be changed at any time.

Windows has two initialization files: WIN.INI and SYSTEM.INI. (The .INI is sometimes pronounced as "Innie".) These .INI files are unformatted text files that are arranged in a specific order. Here's an .INI file structure and an excerpt from an average WIN.INI file:

[Name of Range]
Keyword = Current Value

```
[windows]
BorderWidth = 3
[CorelDraw]
Dir = C:\APPS\CORELDRW
```

This format also applies to your own initialization files. The names used in the ranges (the lines enclosed in [brackets]) must be meaningful. In other words, if a range applies to an application, the application name should be user.

Each range has a series of entries, which specify keywords and the values that apply to the keywords. These values consist of text, integers, or text in quotation marks. Placing text in quotation marks uses the entire text without skipping spaces. A semicolon (;) at the beginning of a line identifies a comment line.

## Standard initialization files

The WIN.INI file contains both application information and system configuration data. The SYSTEM.INI file contains additional system configuration data that controls device drivers. This data cannot be controlled by function calls.

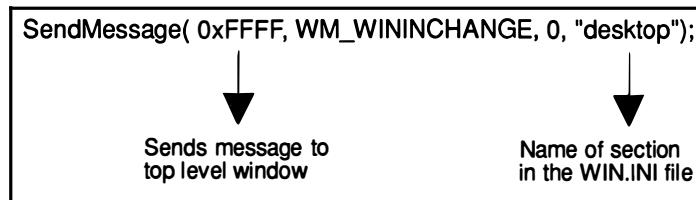
**WIN.INI:** This file can contain up to 11 different system ranges and application specific ranges:

Windows  
Desktop  
Kernel  
Ports  
Colors  
PrinterPorts

Devices  
Fonts  
intl (International)  
Extensions  
PIF

The user can change most of the options listed in this file by using the Windows Control Panel. Changes made from the Control Panel apply to the current Windows session only. Since the WIN.INI file controls all programs, the parameters in WIN.INI address all the applications mentioned in the file. This occurs when the WM\_WININICHANGE messages are sent.

The lParam parameter of this function passes the pointer containing a string that specifies the range name to be changed. If the SendMessage function is assigned the value 0xFFFF instead of a window handle, the message is automatically sent to the top level window.



Three functions enable you to read data from and write data to the WIN.INI file.

```
int GetProfileInt( lpApplicationName, lpKeyName, nDefault)
int GetProfileString( lpApplicationName, lpKeyName, lpDefault, lpReturnedString,
nSize)
BOOL WriteProfileString( lpApplicationName, lpKeyName, lpString)
```

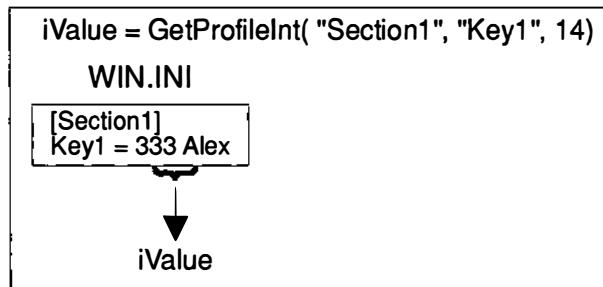
Parameter	Points to
lpApplicationName	Range names, application names
lpKeyName	Keyword names

The GetProfileString function copies a character string from the WIN.INI file to a buffer specified in a previous parameter. The function then searches for the specified range and keyword. When the function finds the keyword, the corresponding string is written to the buffer.

In other instances, the string addressed in the lpDefault variable is copied to the lpReturnedString variable. The search isn't case sensitive (i.e., it doesn't distinguish between uppercase and lowercase characters).

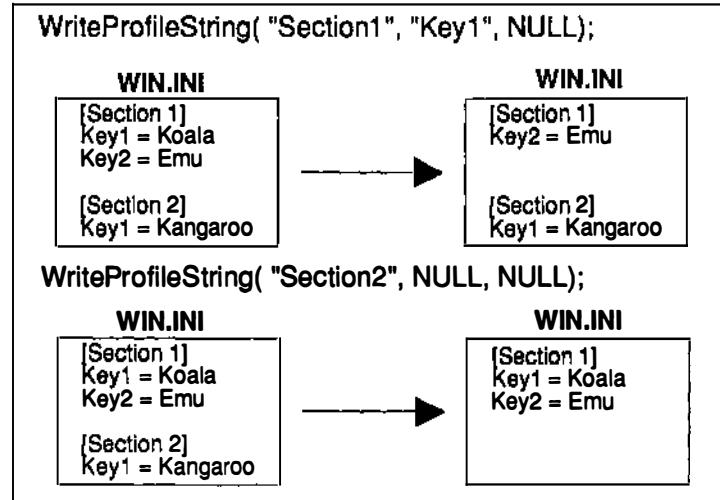
If you discover which keywords are located under a specific range in WIN.INI, the lpKeyName parameter can be set to NULL. The GetProfileString function lists all entries found in the lpReturnedString variable. A "\0" character separates each keyword.

The GetProfileInt function reads numbers from WIN.INI as integer values. If a value associated with a keyword consists of only characters, the function returns a value of zero. If the number precedes a series of characters, only the number will be read.



The WriteProfileString function lets you change WIN.INI from within an application. Because all initialization files are text files, this function cannot contain any integer values. The itoa C function performs a conversion before the last parameter can be passed to Windows.

If a range name or keyword name doesn't exist, WriteProfileString doesn't write the information to WIN.INI. The WriteProfileString function can also delete lines from WIN.INI. If the last parameter contains a value of NULL, the line identifying the keyword is deleted. If the keyword is set to NULL, the function deletes the entire range stated in the first parameter.



**SYSTEM.INI:** This initialization file contains all global system information needed for Windows initialization. The user cannot change these values from the Control Panel. Unlike WIN.INI, there aren't any read and write functions. The SYSTEM.INI file consists of the following four ranges:

## SYSTEM.INI configuration ranges:

- Boot
- Boot.Description
- Win386
- Keyboard

If changes are made to a range (e.g., [Keyboard]), these changes take effect at the start of the next Windows session.

## Creating your own initialization files

Applications can also access user defined initialization files. To create an initialization file, use the following functions:

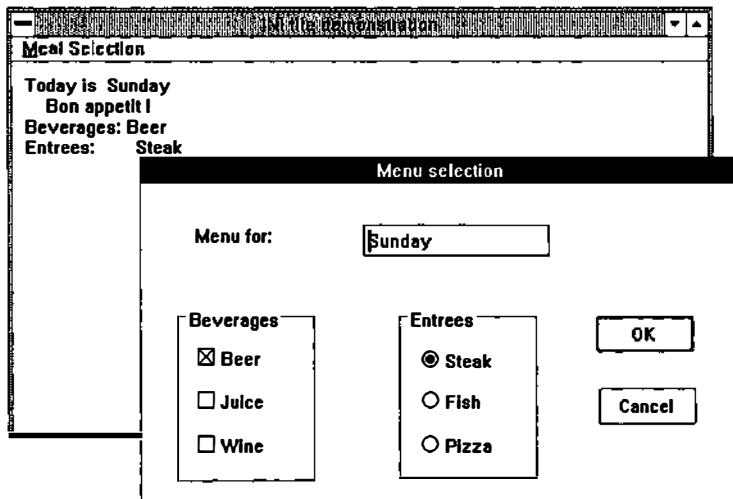
```

int GetPrivateProfileInt( lpApplicationName, lpKeyName, nDefault, lpFileName )
int GetPrivateProfileString( lpApplicationName, lpKeyName, lpDefault,
                           lpReturnedString, nSize, lpFileName )
BOOL WritePrivateProfileString( lpApplicationName, lpKeyName, lpString, lpFileName )
  
```

Parameter	Points to
lpApplicationName	Range names, application names
lpKeyName	Keyword names
lpFileName	Initialization filename

These functions are used the same way as the functions for WIN.INI (see above). However, additional parameters are written to a private initialization file.

## Example of initialization files



The following example creates a private initialization file. The contents of this file appear in the client area. A dialog box enables you to enter the values you want to apply to this .INI file.

The dialog box handles this task as a meal. The user can select one of three entrees and up to three beverages. The menu changes with the day of the week.

### New functions

GetPrivateProfileInt  
GetPrivateProfileString  
GetProfileString  
lstrcpy  
WritePrivateProfileString  
WriteProfileString

### Brief description

Copies numbers from a private .INI file  
Copies data from a private .INI file  
Copies data from WIN.INI  
Appends one string to another  
Writes data to a private .INI file  
Writes data to WIN.INI

## Source code: FINIT.C

```
/** FINIT.C ****
** Demonstrates initialization file creation using a simple menu planning */
** application.
****

#include "windows.h"
#include "finit.h"
#include <string.h>
#include <stdlib.h>

BOOL FinitInit ( HANDLE );
long FAR PASCAL FinitWndProc( HWND, unsigned, WORD, LONG);
BOOL FAR PASCAL MealsDlgProc( HWND, unsigned, WORD, LONG);

char szReturnDay[11];
BOOL bJuice, bWine, bBeer;
char szEntree[6];
WORD wEntree;

int PASCAL WinMain( hInstance, hPrevInstance, lpszCmdLine,cmdShow)
HANDLE hInstance, hPrevInstance;
LPSTR lpszCmdLine;
int cmdShow;
{
    MSG     msg;
    HWND    hWnd;

    if (!hPrevInstance)
    {
        if (!FinitInit( hInstance ))
            return FALSE;
    }
    hWnd = CreateWindow("Finit",
                        "INI file demonstration",
                        WS_OVERLAPPEDWINDOW,
                        CW_USEDEFAULT, 0, CW_USEDEFAULT, 0,
                        NULL, NULL, hInstance, NULL);

    ShowWindow( hWnd, cmdShow );
    UpdateWindow( hWnd );

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

```
    }
    return (int)msg.wParam;
}

BOOL FinitInit( hInstance )
HANDLE hInstance;
{
    WNDCLASS      wcFinitClass;

    wcFinitClass.hCursor      = LoadCursor( NULL, IDC_ARROW );
    wcFinitClass.hIcon        = LoadIcon( NULL, IDI_APPLICATION );
    wcFinitClass.lpszMenuName = (LPSTR)"Meals";
    wcFinitClass.lpszClassName = (LPSTR)"Finit";
    wcFinitClass.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
    wcFinitClass.hInstance     = hInstance;
    wcFinitClass.style        = CS_VREDRAW | CS_HREDRAW;
    wcFinitClass.lpfnWndProc   = FinitWndProc;
    wcFinitClass.cbClsExtra   = 0 ;
    wcFinitClass.cbWndExtra   = 0 ;

    if (!RegisterClass( &wcFinitClass ) )
        return FALSE;
    return TRUE;
}

long FAR PASCAL FinitWndProc( hWnd, message, wParam, lParam )
HWND      hWnd;
unsigned message;
WORD      wParam;
LONG      lParam;
{
    static HANDLE hInst;
    FARPROC      lpprocDia;
    HDC          hDC;
    PAINTSTRUCT ps;
    int          i;

    static char szScrnOutput[4][26] = {"Today is           ",
                                       "      Bon appetit !      ", ",",
                                       "Beverages:           ", ",",
                                       "Entrees:             "};

    static char szSecName1[] = "Date";
    static char szSecName2[] = "Food";
    static char szSecName3[] = "Drink";
    static char szKeyName1[] = "Day";
    static char szKeyName21[] = "Entree";
    static char szKeyName31[] = "Beer";
    static char szKeyName32[] = "Juice";
```

```

static char      szKeyName33[] = "Wine";
static char      szDefaultStr[] = " ";
static char      szReturn[10];
static char      szInitFileName[13];
char            szName[12];
char            lpValue[2];

switch (message)
{
    case WM_CREATE:
        hInst = GetWindowWord(hWnd, GWW_HINSTANCE);

        szDefaultStr[0] = 0;
        GetProfileString("Finit", "MealsInit", szDefaultStr,
                          szInitFileName, 20);
        if (*szInitFileName == 0)
        {
            WriteProfileString("Finit", "MealsInit",
                               "meals.ini");
            lstrcpy( szInitFileName,"meals.ini");
        }

        szDefaultStr[0] = 0;
        GetPrivateProfileString("Init", "Meals",szDefaultStr ,
                               szName, 13, szInitFileName);
        if (lstrcmp(szName, szInitFileName))
        {
            WritePrivateProfileString("Init", "Meals",
                                      "meals.ini", szInitFileName);
            WritePrivateProfileString(szSecName1, szKeyName1,
                                      "Sunday",
                                      szInitFileName);
            WritePrivateProfileString(szSecName2, szKeyName21,
                                      "Steak",szInitFileName);
            WritePrivateProfileString(szSecName3, szKeyName31,
                                      "1",szInitFileName);
            WritePrivateProfileString(szSecName3, szKeyName32,
                                      "0",szInitFileName);
            WritePrivateProfileString(szSecName3, szKeyName33,
                                      "0",szInitFileName);
        }
}

/** Default information for MEALS.INI *****/
GetPrivateProfileString(szSecName1, szKeyName1, "Sunday",
                       szReturnDay, 11,
                       szInitFileName);

```

## File Management

---

```
GetPrivateProfileString(szSecName2, szKeyName21, "Steak",
                        szEntree, 6,
                        szInitFileName);
/** Change entree ****
switch (szEntree[0])
{
    case 'S':
        wEntree = ID_STEAK;
        break;
    case 'F':
        wEntree = ID_FISH;
        break;
    case 'P':
        wEntree = ID_PIZZA;
        break;
    default:
        wEntree = ID_STEAK;
        break;
}

bBeer = GetPrivateProfileInt(szSecName3, szKeyName31, 1,
                             szInitFileName);
bJuice = GetPrivateProfileInt(szSecName3, szKeyName32, 0,
                             szInitFileName);
bWine = GetPrivateProfileInt(szSecName3, szKeyName33, 0,
                             szInitFileName);
break;

case WM_COMMAND:
    switch(wParam)
    {
        case ID_MEALS:
            lpprocDia = MakeProcInstance (MealsDlgProc,
                                         hInst);
            if (DialogBox(hInst, "Meals", hWnd,
                          lpprocDia))
            {
                switch (wEntree)
                {
                    case ID_STEAK:
                        strcpy( szEntree,
                                "Steak");
                        break;
                    case ID_FISH:
                        strcpy( szEntree,
                                "Fish");
                        break;
                }
            }
    }
}
```

```
        case ID_PIZZA:
            lstrcpy( szEntree,
                      "Pizza");
            break;
        default:
            lstrcpy( szEntree,
                      "Steak");
            break;
    }

    WritePrivateProfileString(szSecName1, szKeyName11,
                                szReturnDay, szInitFileName);
    WritePrivateProfileString(szSecName2, szKeyName21,
                                szEntree, szInitFileName);
    itoa(bBeer, lpValue, 10);
    WritePrivateProfileString(szSecName3, szKeyName31,
                                lpValue, szInitFileName);
    itoa(bJuice, lpValue, 10);
    WritePrivateProfileString(szSecName3, szKeyName32,
                                lpValue, szInitFileName);
    itoa(bWine, lpValue, 10);
    WritePrivateProfileString(szSecName3, szKeyName33,
                                lpValue, szInitFileName);

    InvalidateRect (hWnd, NULL, TRUE);

}

FreeProcInstance (lpprocDia);
break;

default:
    break;
}
break;

case WM_PAINT:           // Draw window
hDC = BeginPaint(hWnd, &ps);

lstrcpy(&szScrnOutput[0][10], szReturnDay);
lstrcpy(&szScrnOutput[3][16], szEntree);
i = 0;

/** Display beverage *****/
if (bBeer)
{
    lstrcpy(&szScrnOutput[2][11], "Beer ");
    i = 5;
```

```
        }
        if (bJuice)
        {
            lstrcpy(&szScrnOutput[2][11+i], "Juice ");
            i = i + 6;
        }
        if (bWine)
            lstrcpy(&szScrnOutput[2][11+i], "Wine ");
        if (! (bBeer | bJuice | bWine))
            lstrcpy(&szScrnOutput[2][11], " ");
    }

    for (i=0; i <= 3; i++)
    {
        TextOut(hDC, 10, 10+16*i, &szScrnOutput[i][0],
                 strlen(&szScrnOutput[i][0]));
    }
    EndPaint(hWnd, &ps);
    break;

    case WM_DESTROY:
        PostQuitMessage(0);
        break;

    default:
        return (DefWindowProc( hWnd, message, wParam, lParam ));
        break;
    }
    return(0L);
}

BOOL FAR PASCAL MealsDlgProc( hDlg, message, wParam, lParam )
HWND      hDlg;
unsigned    message;
WORD       wParam;
LONG       lParam;
{

    switch (message)
    {
        case WM_INITDIALOG:
            SetDlgItemText( hDlg, ID_DAY, szReturnDay);
            SetFocus(GetDlgItem(hDlg, ID_DAY));

            SendDlgItemMessage(hDlg,wEntree, BM_SETCHECK, TRUE,
                               (LONG)0);
            SendDlgItemMessage(hDlg, ID_BEER, BM_SETCHECK, bBeer,
                               (LONG)0);
    }
}
```

```
SendDlgItemMessage(hDlg, ID_JUICE, BM_SETCHECK, bJuice,
                   (LONG) 0);
SendDlgItemMessage(hDlg, ID_WINE, BM_SETCHECK, bWine,
                   (LONG) 0);
return(FALSE);
break;

case WM_COMMAND:
switch (wParam)
{
    case ID_STEAK:
    case ID_FISH:
    case ID_PIZZA:
        wEntree = wParam;
        return (TRUE);
        break;

/** Place information in MEALS.INI *****/
case IDOK:
GetDlgItemText( hDlg, ID_DAY, szReturnDay, 11);
bBeer = SendDlgItemMessage(hDlg, ID_BEER,
                           BM_GETCHECK, 0, 0L);
bJuice = SendDlgItemMessage(hDlg, ID_JUICE,
                           BM_GETCHECK, 0, 0L);
bWine = SendDlgItemMessage(hDlg, ID_WINE,
                           BM_GETCHECK, 0, 0L);
EndDialog (hDlg, 1);
return(TRUE);
break;

/** Cancel, do not place information in MEALS.INI *****/
case IDCANCEL:
EndDialog (hDlg, 0);
return(TRUE);
break;

default:
return(FALSE);
}
default:
return(FALSE);
}

}
```

## Module definition file: FINIT.DEF

```
NAME          Finit
DESCRIPTION   'Initialization file demo'
EXETYPE      WINDOWS
STUB          'WINSTUB.EXE'
CODE          PRELOAD MOVEABLE
DATA          PRELOAD MOVEABLE MULTIPLE
HEAPSIZE     4096
STACKSIZE    4096
EXPORTS       FinitWndProc @4
              MealsDlgProc @5
```

## Resource script: FINIT.RC

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

#include "meals.dlg"

Meals MENU
BEGIN
    POPUP  "&Meal Selection"
    BEGIN
        MENUITEM "&Menu Plan",     ID_MEALS
    END
END
```

## Header file: FINIT.H

```
#define ID_MEALS 10
#define ID_BEER   20
#define ID_JUICE  21
#define ID_WINE   22
#define ID_DAY    30
```

```
#define ID_STEAK 40
#define ID_FISH 41
#define ID_PIZZA 42
```

## Dialog box file: MEALS.DLG

```
MEALS DIALOG LOADONCALL MOVEABLE DISCARDABLE 51, 47, 235, 123
CAPTION "Menu selection"
STYLE WS_BORDER | WS_CAPTION | WS_DLGFREAME | WS_POPUP
BEGIN
    CONTROL "Menu for:", -1, "static", SS_LEFT | WS_GROUP |
        WS_CHILD, 21, 16, 65, 9
    CONTROL "", ID_DAY, "edit", ES_LEFT | ES_AUTOHSCROLL | WS_BORDER |
        WS_TABSTOP | WS_CHILD, 87, 16, 73, 12
    CONTROL "Beverages", -1, "button", BS_GROUPBOX | WS_GROUP |
        WS_CHILD, 14, 47, 54, 68
    CONTROL "Beer", ID_BEER, "button", BS_AUTOCHECKBOX | WS_TABSTOP |
        WS_CHILD, 22, 61, 30, 12
    CONTROL "Juice", ID_JUICE, "button", BS_AUTOCHECKBOX |
        WS_CHILD, 22, 78, 28, 12
    CONTROL "Wine", ID_WINE, "button", BS_AUTOCHECKBOX |
        WS_CHILD, 22, 95, 34, 12
    CONTROL "Entrees", -1, "button", BS_GROUPBOX | WS_GROUP |
        WS_CHILD, 101, 47, 54, 67
    CONTROL "Steak", ID_STEAK, "button", BS_AUTORADIOBUTTON | WS_TABSTOP |
        WS_CHILD, 110, 62, 36, 12
    CONTROL "Fish", ID_FISH, "button", BS_AUTORADIOBUTTON |
        WS_CHILD, 110, 78, 34, 12
    CONTROL "Pizza", ID_PIZZA, "button", BS_AUTORADIOBUTTON |
        WS_CHILD, 110, 95, 34, 12
    CONTROL "OK", IDOK, "button", BS_DEFPUSHBUTTON | WS_GROUP | WS_TABSTOP |
        WS_CHILD, 178, 51, 38, 14
    CONTROL "Cancel", IDCANCEL, "button", BS_PUSHBUTTON | WS_TABSTOP |
        WS_CHILD, 179, 79, 38, 14
END
```

To compile and link this application, use the RCOMPILE.BAT described earlier in this chapter. Here's the listing again:

```
cl -c -Gw -Zp %1
rc -r %1.rc
link /align:16 %1,%1.exe,,libw+slibcew,%1.def
rc %1.res
```

Save this file to a directory contained in your path or the directory containing your source, module definition and resource scripts. Type the following and press **Enter** to compile FINIT:

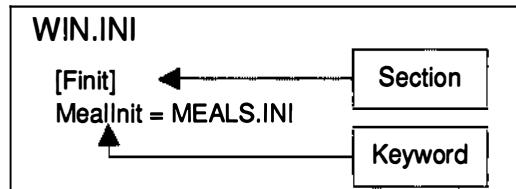
```
rcompile finit
```

## How FINIT.EXE works

The application begins by checking for an existing .INI file listed in WIN.INI. This is done with the GetProfileString function. This filename lies in the [Finit] range under the FoodInit keyword.

If an entry doesn't exist, Windows copies the default string (consisting of "\0") to the szInitFileName return string. In this case, the WriteProfileString function inserts a new range in the WIN.INI file and assigns the name "MEALS.INI" to the keyword. You can use a filename other than MEALS.INI by changing the "Finit" entry.

The following entries are placed in WIN.INI:



The MEALS.INI file consists of four ranges. The first range, [Init], contains the keyword Meals. If this isn't the keyword, something is wrong with the file.

For instance, the file may be empty. If this occurs, re-initialize the file (write all ranges and defaults to this file) by using the WritePrivateProfileString function. The MEALS.INI file should look as follows:

```
[Init]
Meals=meals.ini
[Date]
Day=Sunday
[Food]
Entree=Steak
[Drink]
Beer=1
Juice=0
Wine=0
```

The values in the [Drink] keywords are integer values. We use the WritePrivateProfileString function to insert these values and convert the integer values to ASCII characters in the process.

Regardless of whether the file exists or whether it must be created from scratch, the GetPrivateProfileString and GetPrivateProfileInt functions read the information and write it to a buffer.

The first characters of the entrees indicate the current entree. Each entree and beverage is represented by dialog buttons and ID values. The Entree variable receives the value corresponding to the current entree.

The [Drink] range contains values for each keyword (0 or 1). The GetPrivateProfileInt function returns the integer values written to the corresponding Boolean variable.

Selecting the **Menu Plan** menu item displays the dialog box shown above. This dialog box displays the values taken from the initialization file. The WM\_INITDIALOG message configures buttons and edit controls in the dialog box routine.

When the user selects a new radio button, the button's ID value is noted in the wEntree variable. Clicking on the **OK** button supplies the three Boolean beverage variables with current status. The contents of the edit control are written to the szReturnDay variable.

Once the dialog box closes, the ID values in the wEntree variable indicate whether "Steak", "Fish", or "Pizza" should be copied to the szEntree variable. This text string and the other values are added to the MEALS.INI file by the WritePrivateProfileString function.

The `itoa` function converts the three Beverage Boolean variables into ASCII characters. The `InvalidateRect` function creates a `WM_PAINT` message, which displays the new options.

The resource script file describes the menu with a pop-up menu and one menu item, as well as the dialog box. The `#include` statement adds these features to the application. All dialog box controls are defined in the `FINIT.H` header file as ID values.

# Dynamic Link Libraries

## Introduction

Windows accesses the following Dynamic Link Libraries (DLL):

KERNEL.EXE  
USER.EXE  
GDI.EXE

Windows also has several DLLs that act as device drivers. These drivers have names such as DISPLAY.DRV (screen driver), MOUSE.DRV (mouse support), and SYSTEM.DRV (timer). While static linking permanently combines files (and inflexibly) through the linker, dynamic linking instructs the executable application to call DLLs, which are separate files from the application. The DLLs are called by the IMPORTS function or listed in the Import Library.

Several applications can access the same object from within the same DLL. This object can be a function or a resource. The DLL must be loaded only once in the memory allocated for it. If the application needs an upgrade to functions defined by the DLL, the developer can update, link, and compile only the DLL. The application itself doesn't have to be updated.

The main difference between a Windows application, a task, and a Windows DLL is that a DLL doesn't have a starting address and doesn't react to messages. Instead, it jumps to every function defined in the DLL. A DLL doesn't have a message loop. Usually the tasks in Windows are the active components and the DLL modules wait until they are called.

DLL programming will continue to expand the capabilities of Windows program development. Remember that the DLL must reside in the current directory or be specified in the PATH environment variable.

In this chapter we'll discuss DLLs in detail.

# FAR functions

Because of the segmented memory architecture supplied by the Intel processors, you can divide function calls into segment internal or segment overlapping. If a function is located in the same segment as the code to be called, it is a NEAR function. A function located in a different segment from the code is called a FAR function.

## Basics

A DLL can have its own data segment. So, the DLL must define the proper data segment needed by the DLL or application for calling functions. Since all Windows functions are defined in DLLs (e.g., CreateWindow in the USER.EXE DLL), this problem is always present. The -Gw compiler switch and the MakeProcInstance function solve this problem.

## Prolog and Epilog

When a C compiler creates a C program, the compiler places a code at the beginning and end of each function. These codes, called Prolog and Epilog, allow parameters and local data to access these functions. NEAR functions, which are used only within a Windows application, use Prolog and Epilog.

When the user includes the -Gw switch in the compiler call and defines the Windows application with FAR, the system adds another Prolog. The code then looks similar to the following:

```
Far functions and -Gw switch
PUSH DS
POP AX
NOP
.....
INC BP          Prolog
PUSH BP
MOV BP, SP
PUSH DS
MOV DS, AX
SUB SP, Number
```

The Number value gives the sizes of the local function variable that aren't defined with static. When the function ends, the Epilog receives the register contents, which were placed on the stack by Prolog.

The lines in Prolog manipulate the data segment register. The stack fills the contents of the AX register with the contents of the DS register. Then the BP register writes the contents of the AX register to the data segment register. Now we have the same contents as in the DS register. This is done so that the application calling the function won't terminate.

When this FAR function is entered in the definition file as an EXPORT function, it can be called from Windows. Then, once the segment loads into memory, Windows changes the first two bytes of the Prolog.

```
NOP  
NOP  
NOP  
.....  
INC BP           Prolog  
PUSH BP  
MOV BP, SP  
PUSH DS  
MOV DX, AX  
SUB SP, Number
```

We've replaced the PUSH DS and POP AX instructions used earlier with NOP instructions. The Prolog now stores the contents of the data segment register on the stack and fills this register with the contents of the AX register.

This change ensures that functions exported to an application won't be called directly because the DS register, instead of the application's data segment, contains a random value.

Before we discuss why we made this change, let's look at another use for Prolog. All the functions defined in a DLL consist of FAR functions, the -Gw compiler switch, and the Prolog and Epilog codes mentioned above. When you load the DLL, Windows overwrites the first three bytes in the following way:

```
MOV AX, dlladdress
.....
INC BP          Prolog
PUSH BP
MOV BP, SP
PUSH DS
MOV DX, AX
SUB SP, Number
```

The DLL's data segment address (dlladdress in this example) loads into the AX register. It is possible that every DLL has only one instance, and, therefore, only one data segment. The Prolog instruction stores the data segment on the stack, and loads the DS register with the contents of AX.

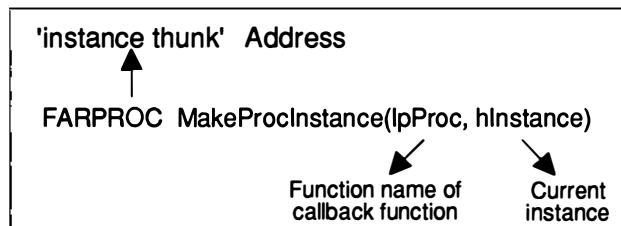
This guarantees that all the functions defined in a DLL access the DLL data instead of the application data.

## Callback functions

The functions defined in a Windows application and called from Windows are callback functions. These functions use the FAR option, and appear in the EXPORTS section of the module definition (.DEF) file. The dialog box is one example of a callback function.

Although a callback function can be enabled from outside Windows, it must be able to use the data segment as the current instance instead of the Windows DLL, which is currently loaded in the data segment register.

Based on the above example containing Prolog and the three NOP instructions, the AX register of the data segment must contain the current instance when the Prolog jumps to the function. The MakeProcInstance function performs this task.

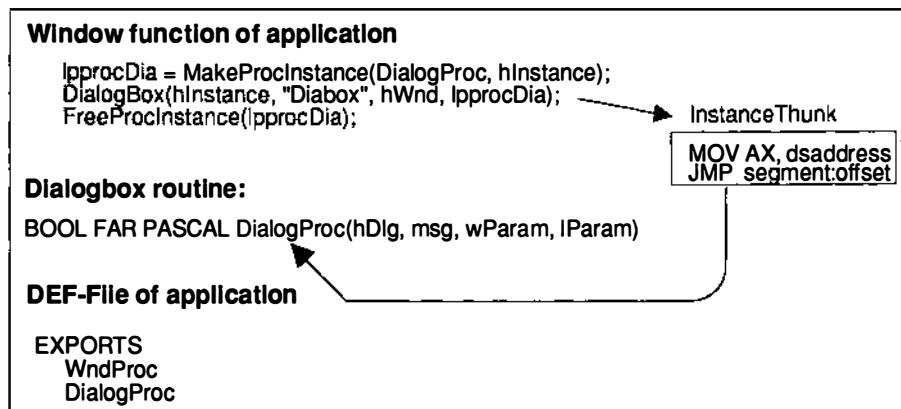


The MakeProcInstance function passes the callback function address and the current instance handle. The given function has an InstanceThunk. This consists of two commands and reads the address as a return value.

```
'instance thunk'
MOV AX, dsaddress
JMP segment:offset
```

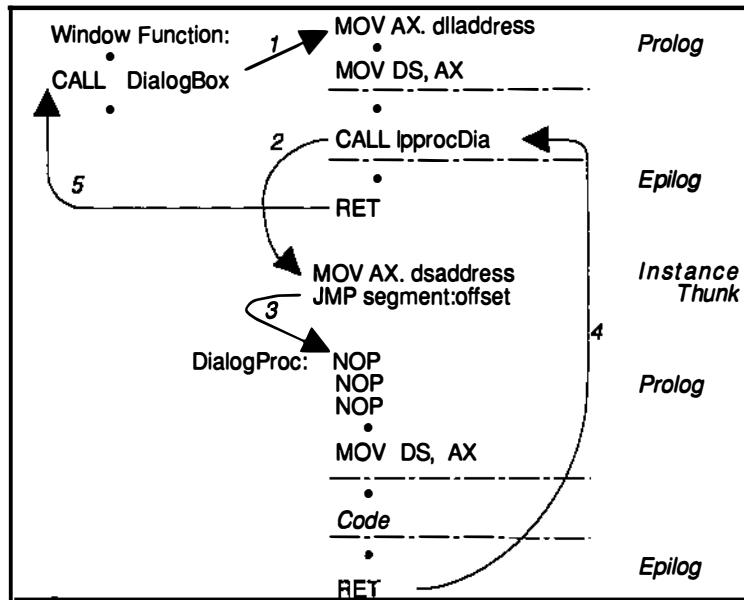
The dsaddress value is the data segment address of the current instance. The address for InstanceThunk is passed to Windows so that Windows can access the callback function properly. After that, the AX register provides the address of the correct data segment and a jump to the Prolog that started the callback function occurs.

The following figure should illustrate this process:



The dialog box routine must be defined with FAR and exported to the module definition (.DEF) file before Windows can call it. The MakeProcInstance function is also needed to create an InstanceThunk. The return value IpprocDia (which points to this two-line code) passes the DialogBox function. When the dialog box routine executes, it branches to the InstanceThunk. From that point, the jump to the dialog box function's Prolog occurs.

The following illustration shows the assembly language for this example.



As soon as a Windows application calls this example, this function branches to the Prolog in the USER.EXE DLL. This Prolog loads the address of the DLL's data segment into the data segment register. When the dialog box routine is called, the data segment of the application must be reloaded into the DS register.

The Prolog for the InstanceThunk and the dialog box routine perform this task. The Thunk loads the address of the data segment's current instance into the AX register. The Prolog then changes the contents of the DS register to equal the contents of the AX register. Now the dialog box function can operate with the correct data segment.

Although the window function used in almost every Windows application must be defined with FAR and the EXPORTS section passed to the module definition file, the InstanceThunk doesn't have to be called by the MakeProcInstance function. This occurs when registering classes.

## Restrictions

Although the DLL has its own data segment, it doesn't have its own stack. The individual DLL functions use the stack that's called for

applications. Once a DLL executes, the data segment (DS) register contents pass to the DLL's data segment while the stack segment (SS) register remains unchanged.

## In a DLL

### In an application

DS != SS DS == SS

**DS = Contents of the data segment register**

**SS** = Contents of the stack segment register

However, the C language always assumes that the contents of the data segment (DS) register and the stack segment (SS) register are identical. This is based on variable addressing in the C language.

All local data (defined with static) and all global data are placed in the application's data segment. The compiler uses near pointers relative to the data segment (DS) register to provide easy access to this data. The stack receives all parameters passed by functions, as well as all local variables not defined as static.

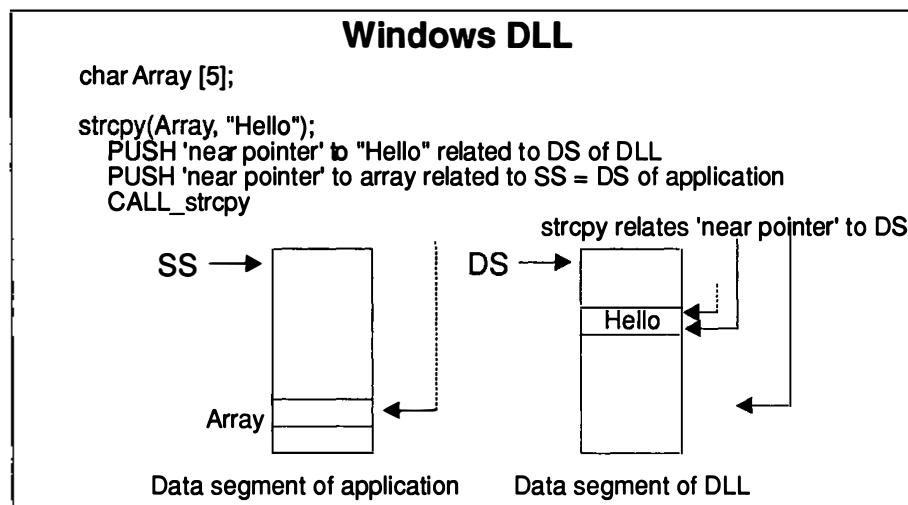
The compiler also uses near pointers here, although this time the stack segment (SS) register handles the data addressing. So, a near pointer is partially relative to the DS register and partially relative to the SS register. Since only the offset is available and not the segment address, the computer has no option for distinguishing which segment covers the near pointer.

Therefore, a C program places the data segment's address in the data segment (DS) register and the stack segment (SS) register.

Now let's look at an example (which uses the `strcpy` C function) that clearly illustrates the theory we just presented. The example should copy the constant string "Hello", which is located in the data segment, to the `Array` variable, which is stored on the stack.

Since the data segment register and stack segment register both contain the data segment address in a Windows application, both data are accessed when the `strcpy` function uses both near pointers relative to the

data segment. However, when a Windows DLL uses this function, undesirable results will occur.



The constant string lies in the DLL's data segment, while the Array variable lies in the stack segment. However, the strcpy function accesses both values through the data segment register, which contains the address of the DLL's data segment. Therefore, the string was copied to the DLL's data segment and not to the Array variable.

When developing DLLs, avoid using any C functions that use near pointers unless these short variables are defined as static or global. Then the DLL's data segment receives the address in the DS register, which allows C functions to operate normally.

Use the Windows specific lstrcpy function instead of the strcpy function. Both of these functions have parameters of data type LPSTR. This data type consists of the start of segment's offset and address. This clarifies the addressing.

In addition to the lstrcpy function, there are four Windows functions that handle far pointers and supersede their normal C counterparts.

lstrcat  
lstrcmp  
lstrcmpi  
lstrcpy  
lstrlen

DLL programming also has another restriction. The DLL function cannot call C functions that have a variable number of passed parameters. After compiling, these variables specify offsets to the stack instead of the entire address. Again, replacing normal C functions with Windows specific functions will solve this problem. For example, you could use the Windows specific wsprintf function instead of the usual sprintf C function.

All parameters that pass data to DLL functions and represent pointers must be far pointers. Windows already includes many of its own data types, such as LPLOGPEN and LPRECT.

## **Creating a DLL**

DLLs can be written to define either functions or resources.

## **Source code**

A DLL can contain an optional initialization routine. This routine executes only when the application calls the DLL for the first time. As other applications, which also access this DLL, start, the calls increment an internal pointer. Any local heap access requires an initialization routine.

For example, you can use this routine to place a starting value in a global variable or to register classes. If you plan on accessing any specific processor registers, you must write part of the routine in assembly language. The Software Development Kit (SDK) contains a sample code named LIBENTRY.ASM.

All functions in a DLL must be defined with FAR. Most of these functions also require the optional PASCAL to ensure shorter and faster code.

Besides the restrictions stated above, DLL functions can be written just like normal Windows applications. Most DLLs will need the WINDOWS.H header file included.

## Resources

The eight standard resource types and user-defined resources can be used as DLLs. Resource DLLs are useful for applications that may be adapted for foreign language markets. The resource DLL is more flexible than a standard resource file. For instance, the DLL can be updated without changing the application.

A resource DLL requires an .RC file similar to other normal .RC files. This file comprises a C file containing the WEP function (see below) and a module definition (.DEF) file. When you compile and link these files using the proper parameters, a resource DLL is created. You can now access this DLL from other applications.

## Wep

All DLLs in Windows must contain an end function named WEP. Windows calls this function when the system ends, or when the DLL's internal counter is set to zero. The WEP function can execute many closing routines, such as closing a database so that the DLL cannot access it. The minimum structure of an end function can look like as follows:

```
VOID FAR PASCAL WEP( int bSystemExit )
{
    if( bSystemExit )
    {
        /* System Shutdown */
    }
    else
    {
        /* DLL counter = 0 */
    }
}
```

The bSystemExit parameter indicates whether the entire system or just the DLL should be shut down. The WEP function must be included in the EXPORTS section of the module definition file. (See the next section for more information.)

## Module definition file

A DLL needs a module definition file. The following is an example of a DLL module definition file:

```
LIBRARY      OwnDLL
DESCRIPTION   'Your own DLL'
EXETYPE      WINDOWS
STUB         WINSTUB.EXE
CODE          PRELOAD MOVEABLE DISCARDABLE
DATA          PRELOAD MOVEABLE SINGLE
HEAPSIZE     0
EXPORTS      WEP @10 RESIDENTNAME
```

The LIBRARY statement indicates that the module is a Dynamic Link Library. The DESCRIPTION, EXETYPE, and STUB statements have the same meanings as those same statements found in an application's module definition file.

The DATA statement can contain either the keyword SINGLE or NONE. This means that the DLL has either one data segment for global and static data or no data segment. You cannot have more than one data segment.

If a DLL uses its local heap, the HEAPSIZE statement specifies this heap's size. The heap must be initialized by a startup function the first time the DLL is called. Frequently the heap size is set to zero.

The STACKSIZE statement assigns the stack size to the application rather than the DLL. In this case, the DLL shouldn't have its own stack.

The EXPORTS section must always be available, and the WEP function must always be available for export. An ordinal number and the RESIDENTNAME statement ensure that the system finds this function as quickly as possible. The RESIDENTNAME statement keeps the export information in memory.

Because of the WEP function, a DLL always contains a code segment that can be specified as either MOVEABLE or DISCARDABLE. Also, all functions that should be called from other modules must also be exported.

## Compiling and linking DLLs

When compiling a DLL, the C compiler needs the -Asnw and -Alnw switches in addition to the usual compiler switches. These switches indicate the memory model that should be assigned to the compiled DLL.

The -Asnw and -Alnw switches activate near pointers for data access. The "s" character assigns the Small memory model, while the "l" assigns the Large memory model.

Both switches inform the compiler that the contents of the data segment and stack segment are identical. Since the compiler cannot assume that the contents are identical, the switches are needed.

When the compiler determines (from an assignment or through a standard C function) that a variable is on the stack through which a near pointer will be accessed, it indicates that further access will be addressed through the data segment.

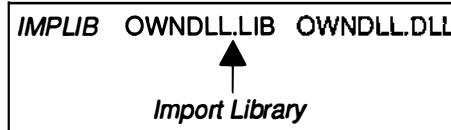
The linker has an additional import library named DLLCEW.LIB. This library contains both a DLL specific startup code and information needed by the routines in the KERNEL.EXE DLL.

After compiling and linking, the resource compiler must be used if the DLL doesn't contain any resources.

RC DLLFCT.DLL

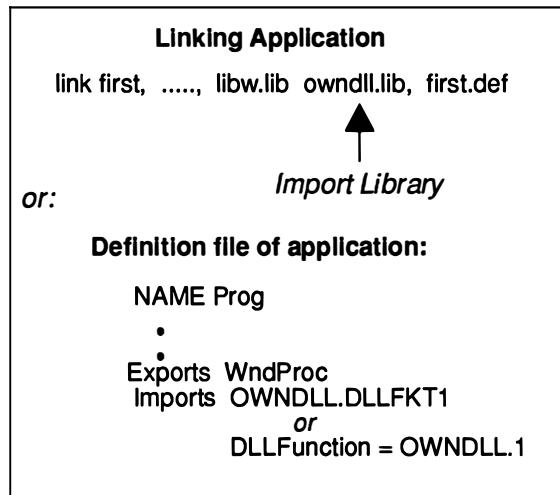
After running the resource compiler, the DLL can be accessed. The IMPLIB application generates an import library. This library indicates the functions contained in the DLL. You can access these functions by linking the import library to applications.

Some DLLs can be compiled and linked as executable files, similar to the GDI.EXE, KERNEL.EXE and USER.EXE DLLs used by Windows. Many of the rules which apply to compiling and linking an executable application also apply to executable DLLs.



## Adding the DLL to an application

In order to use the functions defined in a DLL, these functions must be imported into your application. You can do this either by linking the application with the appropriate import library or by inserting an IMPORTS statement in the module definition file.



If the DLLFCT1 function contains an ordinal number (e.g., 1), you can use the module definition file for importing.

If the DLL accesses other DLLs for resources, these DLLs must also be loaded:

```
HANDLE LoadLibrary( lpLibFileName )
```

Here are two examples. One loads a standard DLL, the other loads a DLL created as an executable file, similar to GDI.EXE:

```
HANDLE hLib;
hLib = LoadLibrary( "OWNRCDLL.DLL" );
```

```
HANDLE hLib;
hLib = LoadLibrary( "OWNRCDLL.EXE" );
```

The returned handle indicates the module instance that's loaded. If the load process fails, the return parameter contains a value that is less than 32. The valid handle is used by all subsequent functions that load a resource.

Here's a call to a resource defined in an .RC resource script, for an application not requiring a DLL:

```
hAccel = LoadAccelerator( hInstance, "AccelTable" );
```

Here's the same call to the same resource, with this resource defined in DLL:

```
hAccel = LoadAccelerator( hLib, "AccelTable" );
```

The FreeLibrary function frees the library when the application ends.

The LoadLibrary function can also be used to link an application with a library module during runtime. If you do this, an IMPORTS statement in the module definition file or an import library statement set isn't needed.

This is illustrated in the following example. The OWNDLL.DLL file defines the TreeOutput function. This function is exported to the module definition file with the ordinal number 5. To call this function, the following lines must be implemented in a Windows application.

```
HANDLE hLib;
FARPROC lpfnTreeO;

hLib = LoadLibrary( "OWNDLL.DLL" );

lpfnTreeO = GetProcAddress( hLib, MAKEINTRESOURCE( 5 ) );

OR

lpfnTreeO = GetProcAddress( hLib, "TreeOutput" );

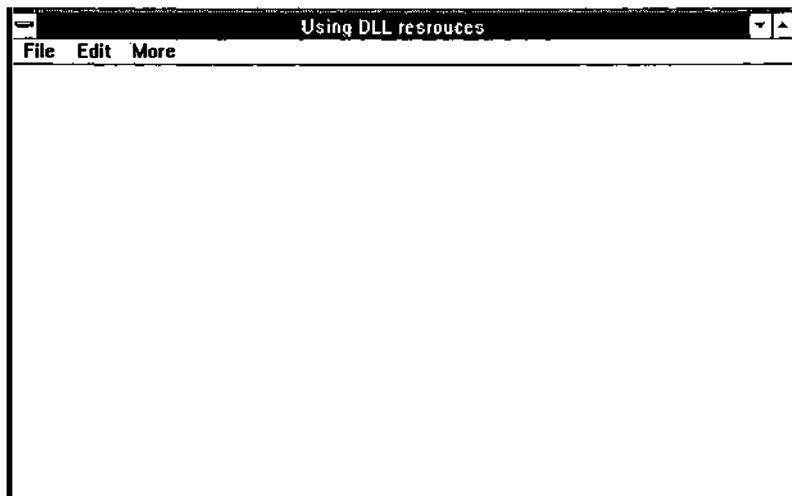
(*lpfnTreeO)();
FreeLibrary( hLib );
```

First the DLL must be loaded using the `GetProcAddress` function, by the ordinal number or the desired function's address name. After the function executes, the `FreeLibrary` function must free the library and decrement the internal DLL counter.

## DLL examples

We've included two examples of DLL access. One demonstrates how a DLL can be used as a global resource file and the other demonstrates how a DLL can be used for passing a group of functions.

### Example of DLL resource access



This example contains its own cursor and an icon (for minimized status). You can create an icon by using the `SDKPAINT.EXE` application found in the Windows Software Development Kit (SDK).

The menu bar contains three menu titles. Selecting the **Append** menu item from the **More** menu, or pressing **Ctrl + A**, takes a new menu item name from a string table and adds this new item to the **More** menu. All the resources for this example are stored in a resource DLL, which is compiled as an executable file, similar to `KERNEL.EXE`.

The **File** and **Edit** menus and their respective menu items are dummies.

We'll list the DLL information first, along with compiling and linking instructions, then we'll list the application.

New functions	Brief description
FreeLibrary	Decrements the DLL counter and removes the DLL from memory when the counter equals 0
LoadLibrary	Loads a library module

## DLL source code: DLLRC.C

```
/** DLLRC.C ****
** Source code for DLLRC.EXE dynamic link library ****
****

#include "windows.h"                                // Include windows.h header file

VOID FAR PASCAL WEP (int);                         // Declare WEP

VOID FAR PASCAL WEP (int bSystemExit)
{
    if (bSystemExit)                                 // Error?
    {
        MessageBeep(0);                            // Then
        MessageBeep(0);                            // -BEEP-
                                                // twice
                                                // System Shutdown
    }
    else                                         // Everything OK?
    {
        MessageBeep(0);                            // Then
                                                // beep once
                                                // DLL counter = 0
    }
}
```

## DLL module definition file: DLLRC.DEF

```
LIBRARY      DLLRC

DESCRIPTION 'DLL with resources'

EXETYPE     WINDOWS

STUB         'WINSTUB.EXE'
```

---

```

CODE           PRELOAD MOVEABLE DISCARDABLE
DATA          PRELOAD NONE

HEAPSIZE      0

EXPORTS       WEP @1 RESIDENTNAME

```

## DLL resource script: DLLRC.RC

```

#include "windows.h"
#include "dll.h"

house      ICON      house.ico
pointer    CURSOR   pointer.cur

DLLMenu MENU
BEGIN
    POPUP  "File"
    BEGIN
        MENUITEM      "New", ID_NEW
        MENUITEM      "Open", ID_OPEN
        MENUITEM      "Exit", ID_EXIT
    END

    POPUP  "Edit"
    BEGIN
        MENUITEM      "Cut",     ID_CUT
        MENUITEM      "Copy",    ID_COPY
        MENUITEM      "Paste",   ID_PASTE
        MENUITEM      "Delete",  ID_DEL
    END
    POPUP  "More"
    BEGIN
        MENUITEM      "Append\tCtrl+A", ID_APP
    END
END

DLLAccel ACCELERATORS
BEGIN
    "^A", ID_APP
END

STRINGTABLE
BEGIN
    STR1,   "America"
    STR2,   "Africa"

```

```
STR3,      "Australia"  
STR4,      "Asia"  
STR5,      "Europe"  
END
```

## Header file: DLL.H

```
#define ID_NEW    20  
#define ID_OPEN   21  
#define ID_EXIT   22  
#define ID_CUT    23  
#define ID_COPY   24  
#define ID_PASTE  25  
#define ID_DEL    26  
#define ID_APP    27  
  
#define STR1     10  
#define STR2     11  
#define STR3     12  
#define STR4     13  
#define STR5     14  
#define STR6     15
```

To compile and link this DLL, we created a batch file named DLLRCBAT.BAT. This batch file is much more complex than batch files listed earlier. When you run DLLRCBAT.BAT, it compiles and links DLLRC as an executable file. Here's the listing for DLLRCBAT.BAT:

```
cl -c -Asnw -W2 -Gw -Zp -batch -Od -Zi -Gi DLLRC.MDT -Gs -FoDLLRC.OBJ DLLRC.C  
rc -r dllrc.rc  
link /NOD /NOE /CO /INC DLLRC,,NUL,slibce+sdl1cew+libw,dllrc.def  
rc dllrc.res
```

Create the batch file (or use the one in the BATCHES directory on the companion diskette for this book). Type the following and press **[Enter]** to create DLLRC.EXE:

```
DLLRCBAT DLLRC
```

## Application source code: FIRSTDRC.C

```
/** FIRSTDRC.C ****
** Improved version of FIRST.C application described in second chapter.  */
** This application accesses external resources for information.      */
*****
```

```
#include "windows.h"                                // Include windows.h header file
#include "dll.h"                                    // Include dll.h header file

HANDLE hLib;                                       // Handle to library

BOOL FirstInit ( HANDLE );
long FAR PASCAL FirstWndProc( HWND, unsigned, WORD, LONG);

/** WinMain (main function for every Windows application) ****

int PASCAL WinMain( hInstance, hPrevInstance, lpszCmdLine, cmdShow )
HANDLE hInstance, hPrevInstance;                  // Current & previous instances
LPSTR lpszCmdLine;                            // Long ptr to string after
                                              // program name during execution
int cmdShow;                                  // Specifies the application
                                              // window's appearance
{
    MSG msg;                                     // Message variable
    HWND hWnd;                                    // Window handle
    HANDLE hAccel;                             // Accelerator handle

    hLib = LoadLibrary("dllrc.exe");           // Load executable file as DLL
    if (hLib < 32)                            // If DLL cannot be loaded
        return FALSE;                          // return FALSE

    if (!hPrevInstance)                         // Initialize first instance
    {
        if (!FirstInit( hInstance )) // If initialization fails
            return FALSE;                      // return FALSE
    }
/** Specify appearance of application's main window ****

hWnd = CreateWindow("First",                           // Window class name
                    "Using DLL resources",          // Window caption
                    WS_OVERLAPPEDWINDOW,          // Overlapped window style
                    CW_USEDEFAULT,                // Default upper-left x pos.
                    0,                           // Upper-left y pos.
                    CW_USEDEFAULT,                // Default initial x size
                    0,                           // y size
                    NULL,                        // No parent window
```

## Dynamic Link Libraries

---

```
        LoadMenu(hLib, "DLLMenu"),           // Window menu used
        hInstance,                         // Application instance
        NULL);                            // No creation parameters

    ShowWindow( hWnd, cmdShow );         // Make window visible
    UpdateWindow( hWnd );               // Update window

    hAccel = LoadAccelerators(hLib, "DLLAccel"); // Get DLL accelerators

    while (GetMessage(&msg, NULL, 0, 0)) // Message reading
    {

/** Accelerator translation *****/
        if (!TranslateAccelerator(hWnd, hAccel, &msg))
        {
            TranslateMessage(&msg);          // Message translation
            DispatchMessage(&msg);          // Send message to Windows
        }
    }

    return (int)msg.wParam;             // Return wParam of last message
}

BOOL FirstInit( hInstance )           // Initialize instance handle
HANDLE hInstance;                   // Instance handle
{
    /* Specify window class *****/
    WNDCLASS      wcFirstClass;          // Main window class

    wcFirstClass.hCursor     = LoadCursor( hLib, "pointer" );
                                // Mouse cursor
    wcFirstClass.hIcon       = LoadIcon( hLib, "house" );
                                // Default icon
    wcFirstClass.lpszMenuName = NULL;
                                // No menu
    wcFirstClass.lpszClassName = "First";
                                // Window class
    wcFirstClass.hbrBackground = GetStockObject( WHITE_BRUSH );
                                // White background
    wcFirstClass.hInstance   = hInstance;        // Instance
    wcFirstClass.style      = 0;                // Horizontal & vertical redraw
                                                // of client area
    wcFirstClass.lpfnWndProc = FirstWndProc;    // Window function
    wcFirstClass.cbClsExtra  = 0 ;              // No extra bytes
    wcFirstClass.cbWndExtra  = 0 ;              // No extra bytes

    if (!RegisterClass( &wcFirstClass ) )      // Register window class

```

```
        return FALSE;           // Return FALSE if registration fails
                                // If registration is successful,
        return TRUE;           // Return TRUE
}
/** FirstWndProc ****
/** Main window function: All messages are sent to this window      */
/** ****
long FAR PASCAL FirstWndProc( hWnd, message, wParam, lParam )
HWND    hWnd;           // Window handle
unsigned message;       // Message type
WORD    wParam;         // Message-dependent 16 bit value
LONG    lParam;         // Message-dependent 32 bit value
{
    static int i = STR1;
    static HMENU hSubMenu, hMenu;
    char szString[12];

    switch (message)           // Process messages
    {
        case WM_CREATE:       // Create window
            hMenu = GetMenu(hWnd);
            hSubMenu = GetSubMenu(hMenu, 2);
            break;

        case WM_COMMAND:       // Add to More menu
            if (wParam == ID_APP)
            {
                LoadString(hLib, i, (LPSTR)szString, 12);
                AppendMenu(hSubMenu, MF_ENABLED, i, szString);
                i++;               // Add a string to this menu
                if (i > STR5)     // i greater than 5?
                    i = STR1; // Append STR1
            }
            break;

        case WM_DESTROY:        // Send WM_QUIT
            FreeLibrary(hLib); // Release library
            PostQuitMessage(0);
            break;              // End of this message process

        default:                // Send other messages to
                                // default window function
            return (DefWindowProc( hWnd, message, wParam, lParam ));
            break;              // End of this message process
    }
    return(0L);
}
```

## Application module definition file: FIRSTDRC.DEF

NAME	FirstDrc
DESCRIPTION	'1st Windows Application with DLL - resource capability'
EXETYPE	WINDOWS
STUB	'WINSTUB.EXE'
CODE	PRELOAD MOVEABLE DISCARDABLE
DATA	PRELOAD MOVEABLE MULTIPLE
HEAPSIZE	4096
STACKSIZE	4096
EXPORTS	FirstWndProc @1

To compile and link this application, and add the DLL to the application, we created a batch file named COMP\_RC.BAT. Here's the listing for this file:

```
cl -c -AS -Gw -Zp %1.c
link /align:16 %1,%1.exe,,slibcew+libw+%2,%1.def
rc %1.exe
```

Type the following and press **Enter** to create the application:

```
comp_rc firstdrc dllrc
```

## How FIRSTDRC.EXE and DLLRC.EXE work

The DLL's .RC file contains information about the cursor and icon, a string table containing five strings, an accelerator table with the **Ctrl** + **A** keyboard accelerator, and a menu consisting of three submenus.

The source text requires the WEP function. The MessageBeep function instructs the computer to beep each time the DLL counter decrements to zero. When the user exits the Windows system, the WEP function goes to the IF branch and the computer beeps twice.

In the module definition file, the LIBRARY keyword instructs the linker to link a DLL. Since the resource DLL doesn't contain any of its own

data, the DATA instruction contains the word NONE. The WEP function must be exported so that Windows can access the DLL.

Both the .RC file and the source code include the DLL.H header file, while both files use the DLL resources. This header file defines ID values for menu items and strings.

We adapted the FIRSTDRC.EXE application from the FIRST.EXE application, which was presented at the beginning of this book, to the DLL. The library handle is stored as global to allow access to the WinMain function and the window functions. The LoadLibrary function loads the DLLRC.DLL file at the beginning of the application.

If the DLL isn't in the current directory or in a directory defined by PATH, Windows displays a message box that instructs the user to insert the diskette, containing the DLL, in drive A:. The user can click a button to retry or cancel the access. If the user cancels, the DLL handle contains an error code (i.e., a value less than 32) and the application ends.

The WNDCLASS structure loads the cursor and icon declared by the DLL handle (instead of the instance handle). The third from last parameter of the DLLMenu menu handle passes data for window creation. This information is obtained from the DLL through the LoadMenu function. The DLL handle for loading the accelerator table is also loaded.

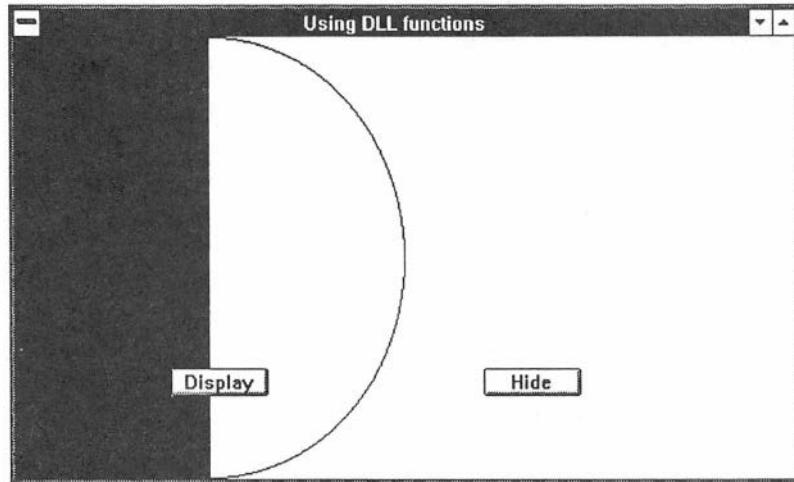
The window function sends three messages: WM\_CREATE, WM\_COMMAND, and WM\_DESTROY. The WM\_CREATE message provides the menu handle for the **More** menu. This menu handle is used by the WM\_COMMAND message.

Here you'll find the ID value for the **Append** menu item. Whenever this item is selected, a variable receives a string from the string table defined in the DLL's resource file. The string can then be added to the **More** menu by selecting the **Append** menu item. When the end of the string table occurs, the counter resets to its starting value.

Before the application itself can close, the internal counter must be decremented using the FreeLibrary function. If this counter is set to

zero, Windows removes the DLL from memory, if no other applications are currently using this DLL.

## Example of DLL function access



The FIRST.EXE application uses a function located in the DLLFUNC.DLL file for generating a child window of type PUSHBUTTON. The buttons allow the user to draw and delete an ellipse and a rectangle.

The rectangle can be filled with either a hatch or solid brush, depending on the color selected. These colors are sent to the application by a callback function. An additional DLL function controls the display of these graphic figures.

We'll list the DLL information first, then the application information.

### New functions

FillRect  
GetSysColor  
MAKELONG

### Brief description

Fills the rectangle with a brush  
Provides color for window sections  
Creates a variable of type DWORD

## DLL source code: DLLFUNC.C

```
/** DLLFUNC.C ****
/** Source code for DLL, supplying functions to FIRSTDFC.EXE application **/
/** ****

#include "windows.h"                                // Include windows.h header file
#include "dllfunc.h"                                // Include dllfunc.h header file

VOID FAR PASCAL WEP (int);                         // Declare WEP

VOID FAR PASCAL WEP (int bSystemExit)
{
    if (bSystemExit)                                // Error?
    {
        .                                         // System Shutdown
    }
    else
    {
        .                                         // Everything OK?
        // Then exit
        // DLL counter = 0
    }
}

/** Create push button ****

DWORD FAR PASCAL CreatePushB (HWND hWndParent, LPSTR lpButtonName)
{
HANDLE hInst;
HWND     hButton;
DWORD   dButton;
static WORD idButton =0;

    idButton++;
    hInst = GetWindowWord(hWndParent, GWW_HINSTANCE);
    hButton = CreateWindow("button", lpButtonName,
                          BS_PUSHBUTTON | WS_VISIBLE | WS_CHILD, 0,0,0,0,
                          hWndParent, idButton, hInst, NULL);
    dButton = MAKELONG(hButton, idButton);
    return (dButton);
}

/** Specify screen ****

BOOL FAR PASCAL Screen(HDC hDC, POINT ptRec, POINT ptEll, FARPROC lpCallScreen)
{
int iColorIndex;
HBRUSH hBrush;
```

```
RECT      rect;

Ellipse(hDC, 0,0, ptEll.x, ptEll.y);

iColorIndex = (* lpCallScreen)();

switch (iColorIndex)
{
    case 1:
        hBrush = CreateHatchBrush(HS_HORIZONTAL, 0x00FF0000);
        break;
    case 2:
        hBrush = CreateHatchBrush(HS_CROSS, 0x0000FF00);
        break;
    case 3:
        hBrush = CreateHatchBrush(HS_VERTICAL, 0x000000FF);
        break;
    default:
        hBrush = CreateSolidBrush(0x00000000);
        break;
}
rect.left = 0;
rect.top = 0;
rect.right = ptRec.x;
rect.bottom = ptRec.y;
FillRect(hDC, &rect, hBrush);
DeleteObject(hBrush);
return (TRUE);
}
```

## DLL module definition file: DLLFUNC.DEF

```
LIBRARY      DLLFUNC

DESCRIPTION  'DLL with functions'

EXETYPE     WINDOWS

STUB         'WINSTUB.EXE'

CODE         PRELOAD MOVEABLE DISCARDABLE
DATA         PRELOAD MOVEABLE SINGLE

HEAPSIZE    0

EXPORTS     CreatePushB @1
```

```
Screen      @2
WEP        @3 RESIDENTNAME
```

## DLL header file: DLLFUNC.H

```
DWORD FAR PASCAL CreatePushB (HWND, LPSTR);
BOOL FAR PASCAL Screen(HDC, POINT, POINT, FARPROC);
```

To compile and link this DLL we created a batch file named DCOMPILE.BAT. This file compiles the source code and generates a library file as well as a DLL. Here's the listing:

```
cl -Asnw -c -Gw -Zp %1.c
link /align:16 %1,%1.dll,,sdllcew+libw,%1.def
rc %1.dll
implib %1.lib %1.dll
```

Type the following and press **Enter** to compile and link DLLFUNC.DLL:

```
dcompile dllfunc
```

## Application source code: FIRSTDFC.C

```
/** FIRSTDFC.C ****
/** Improved version of FIRST.C application described in second chapter.  */
/** This application accesses a DLL function for outside information.  */
/** ****

#include "windows.h"                                // Include windows.h header file
#include "dllfunc.h"                                 // Include dllfunc.h header file

BOOL FirstInit ( HANDLE );
int FAR PASCAL ScreenColor(VOID);
long FAR PASCAL FirstWndProc( HWND, unsigned, WORD, LONG);

/** WinMain (main function for every Windows application) ****

int PASCAL WinMain( hInstance, hPrevInstance, lpszCmdLine, cmdShow )
HANDLE  hInstance, hPrevInstance;                  // Current & previous instances
LPSTR   lpszCmdLine;                            // Long ptr to string after
                                                // program name during execution
int     cmdShow;                                // Specifies the application
                                                // window's appearance
```

```
{  
    MSG      msg;                      // Message variable  
    HWND     hWnd;                     // Window handle  
  
    if (!hPrevInstance)                // Initialize first instance  
    {  
        if (!FirstInit( hInstance )) // If initialization fails  
            return FALSE;           // return FALSE  
    }  
/** Specify appearance of application's main window ******/  
  
    hWnd = CreateWindow("First",                         // Window class name  
                       "Using DLL functions",          // Window caption  
                       WS_OVERLAPPEDWINDOW,          // Overlapped window style  
                       CW_USEDEFAULT,               // Default upper-left x pos.  
                       0,                           // Upper-left y pos.  
                       CW_USEDEFAULT,               // Default initial x size  
                       0,                           // y size  
                       NULL,                        // No parent window  
                       NULL,                        // Window menu used  
                       hInstance,                  // Application instance  
                       NULL);                     // No creation parameters  
  
    ShowWindow( hWnd, cmdShow );                    // Make window visible  
    UpdateWindow( hWnd );                          // Update window  
  
    while (GetMessage(&msg, NULL, 0, 0)) // Message reading  
    {  
  
        TranslateMessage(&msg);           // Message translation  
        DispatchMessage(&msg);           // Send message to Windows  
    }  
  
    return (int)msg.wParam;                         // Return wParam of last message  
}  
  
BOOL FirstInit( hInstance )                      // Initialize instance handle  
HANDLE hInstance;                            // Instance handle  
{  
  
/** Specify window class ******/  
  
    WNDCLASS      wcFirstClass;                 // Main window class  
  
    wcFirstClass.hCursor      = LoadCursor( NULL, IDC_ARROW );  
                                // Mouse cursor  
    wcFirstClass.hIcon       = LoadIcon( NULL, IDI_APPLICATION );  
                                // Default icon
```

```

wcFirstClass.lpszMenuName = NULL;                                // No menu
wcFirstClass.lpszClassName = "First";                             // Window class
wcFirstClass.hbrBackground = GetStockObject( WHITE_BRUSH );      // White background
wcFirstClass.hInstance     = hInstance;                            // Instance
wcFirstClass.style        = CS_HREDRAW | CS_VREDRAW;           // Horizontal
                                         // & vertical redraw
                                         // of client area
wcFirstClass.lpfnWndProc  = FirstWndProc;                         // Window function
wcFirstClass.cbClsExtra   = 0 ;                                    // No extra bytes
wcFirstClass.cbWndExtra   = 0 ;                                    // No extra bytes

if (!RegisterClass( &wcFirstClass ) )          // Register window class
    return FALSE;                                     // Return FALSE if registration fails
                                                    // If registration is successful,
return TRUE;                                              // Return TRUE
}

/** FirstWndProc ****
/** Main window function: All messages are sent to this window      */
/* ****
long FAR PASCAL FirstWndProc( hWnd, message, wParam, lParam )
HWND      hWnd;                                         // Window handle
unsigned  message;                                      // Message type
WORD      wParam;                                       // Message-dependent 16 bit value
LONG      lParam;                                       // Message-dependent 32 bit value
{
    static HANDLE      hInst;
    static DWORD       dButton1, dButton2;
    static BOOL        bScreen = FALSE;
    static WORD        xClient, yClient;
    POINT             pt1,pt2;
    HDC               hDC;
    PAINTSTRUCT       ps;
    FARPROC           lpCallScreenColor;

    switch (message)                                     // Process messages
    {
        case WM_CREATE:                               // Create window
            hInst = GetWindowWord(hWnd, GWW_HINSTANCE);
            dButton1 = CreatePushB(hWnd, (LPSTR) "Display");
            dButton2 = CreatePushB(hWnd, (LPSTR) "Hide");
            break;

        case WM_SIZE:
            xClient = LOWORD(lParam);
            yClient = HIWORD(lParam);
            MoveWindow(LOWORD(dButton1), xClient/5, 3*yClient/4,

```

```
    70, 20, TRUE);
MoveWindow(LOWORD(dButton2), 3*xClient/5, 3*yClient/4,
           70, 20, TRUE);
break;

case WM_COMMAND:
    if (wParam == HIWORD(dButton1))
    {
        InvalidateRect(hWnd, NULL, TRUE);
        bScreen = TRUE;
    }
    else
    {
        if (wParam == HIWORD(dButton2))
        {
            InvalidateRect(hWnd, NULL, TRUE);
            bScreen = FALSE;
        }
    }
break;

case WM_PAINT:
    hDC = BeginPaint(hWnd, &ps);
    if (bScreen)
    {
        pt1.x = xClient/4;
        pt1.y = yClient;
        pt2.x = xClient/2;
        pt2.y = yClient;

        lpCallScreenColor =
            MakeProcInstance(ScreenColor, hInst);
        Screen(hDC, pt1, pt2, lpCallScreenColor);
        FreeProcInstance(lpCallScreenColor);
    }
    EndPaint(hWnd, &ps);
break;

case WM_DESTROY:           // Send WM_QUIT
    PostQuitMessage(0);
break;                      // End of this message process

default:                   // Send other messages to
                           // default window function
    return (DefWindowProc( hWnd, message, wParam, lParam ));
break;                      // End of this message process
}

return(0L);
```

```
}

int FAR PASCAL ScreenColor()
{
    DWORD dColor;

    dColor = GetSysColor(COLOR_ACTIVECAPTION);
    switch(dColor)
    {
        case (0x00FF0000):
            return 1;

        case (0x0000FF00):
            return 2;

        case (0x000000FF):
            return 3;

        default:
            return 0;
    }
}
```

## Application module definition file: FIRSTDFC.DEF

NAME	FirstDfc
DESCRIPTION	'1st Windows Application with DLL functions'
EXETYPE	WINDOWS
STUB	'WINSTUB.EXE'
CODE	PRELOAD MOVEABLE
DATA	PRELOAD MOVEABLE MULTIPLE
HEAPSIZE	4096
STACKSIZE	4096
EXPORTS	FirstWndProc @1 ScreenColor @2

To compile and link this application and the DLL, we created a batch file named COMP\_FNC.BAT. Here's the listing:

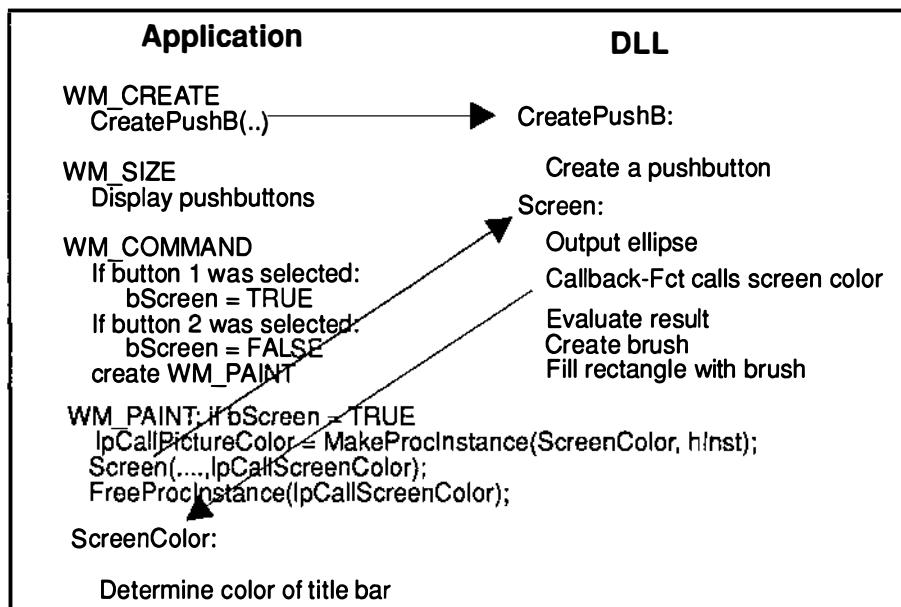
```
cl -c -AS -Gw -Zp %1.c  
link /align:16 %1,%1.exe,,slibcew+libw+%2,%1.def  
rc %1.exe
```

Type the following and press **Enter** to create FIRSTDFC.EXE:

```
comp_fnc firstdfc dllfunc
```

## How FIRSTDFC.EXE and DLLFUNC.DLL work

When the application calls the CreatePushB and Screen DLL functions, the following appears:



The WM\_CREATE message creates the buttons. CreatePushB must pass the main window handle and text, which are initialized to a size of zero. Since every child window must have its own ID value for later use, a counter named idButton increments on every function call. After the instance handle passes, the normal CreateWindow function creates a button. The ID value and the child window's handle are returned to a variable of type DWORD.

The window handle uses the MoveWindow function, which calls the WM\_SIZE message, for providing a window size and making the window visible.

As soon as the user clicks on a button, a WM\_COMMAND message is sent to its parent window. The wParam parameter contains the ID value of this button. If the user clicks the **Hide** button, the Boolean variable bScreen is set to the value FALSE. (In any other case, this variable is set to TRUE.) The InvalidateRect function creates a WM\_PAINT message.

This message reads the set bScreen switch. If the variable is TRUE, the figure appears in the upper-left corner of the client area. The Screen DLL function has four parameters: A handle to the display context and two variables of type POINT, which specify the sizes of the figures. The client area's points of origin are the starting coordinates for the figures. The last parameter is the address of the Instance Thunks as passed by the ScreenColor callback function. This must be determined by the MakeProcInstance function.

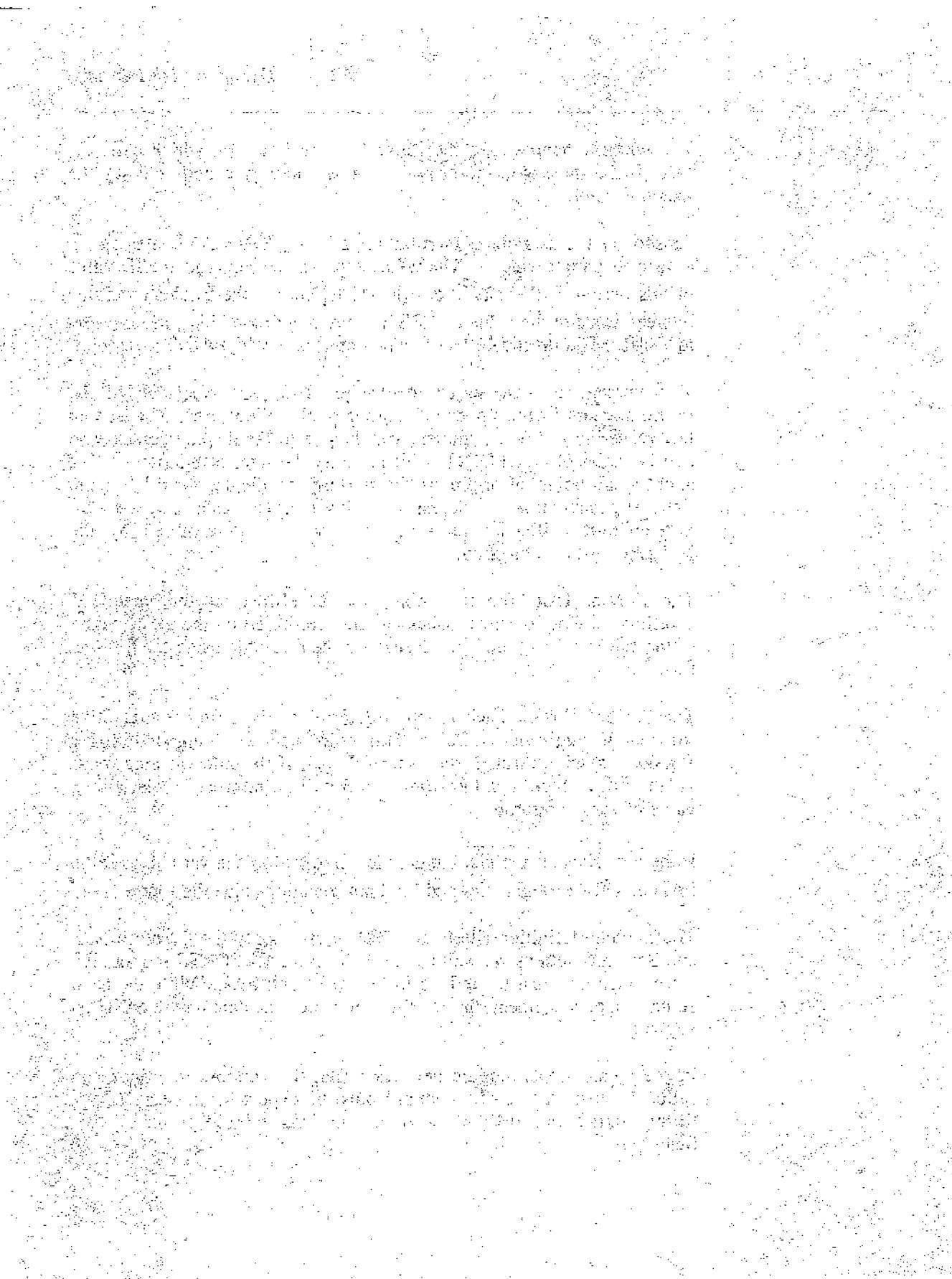
The Screen DLL function displays the ellipse at the specified coordinates. One to four brushes are created, based on the return value of the callback function. The brushes fill the rectangle using the FillRect function.

The ScreenColor callback function determines the color of the active title bar through GetSysColor. This color can be changed during a Windows session through the Control Panel. If the title bar appears as green, red, or blue, a value from 1 to 3 will be returned. Otherwise, a value of zero is returned.

If the user clicks the **Hide** button, the bScreen Boolean variable is set to FALSE and the BeginPaint and EndPaint functions hide the figure.

The module definition file of the DLL must supply the CreateBrushB and Screen functions, as well as the WEP in the EXPORTS section. All three functions must be defined in the DLL with FAR. While the DLL contains a data segment, the DATA statement must contain the SINGLE keyword.

In the application's module definition file, the window functions and callback function (a FAR function) must be exported. Since an import library exists, an IMPORTS statement isn't needed for the DLL functions.

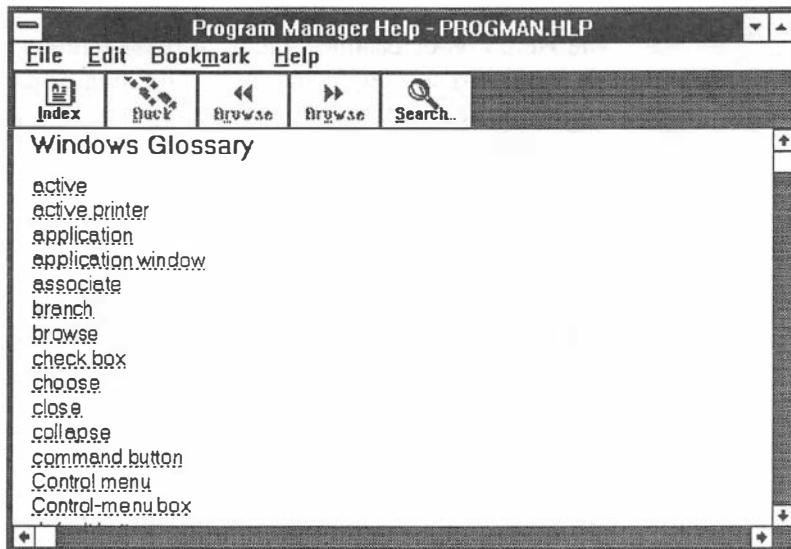


# The Help System

## Overview

The Help system gives the user online information about the application currently running. Since this online Help is a vital part of making applications as user-friendly as possible, many Windows applications include help systems based on the Windows Help application. To create such a Help system, two steps are necessary. The information must be collated, edited, and processed, and the application must be programmed appropriately.

Most Windows applications activate the Help application from a menu named Help.



This menu opens a new window that lists the help text. This window always has the same appearance - only the contents of the client area can be edited.

Like any window, the Help window can be closed using the System menu. After the application ends, the Help window is removed from the screen.

# Help application

The Help application WINHELP.EXE is packaged with Windows. As soon as the user selects a Help menu or item, the Help window appears on the screen. The five buttons at the top of the screen let the user select the options that can be used in searching for help.

These buttons are:

[Index] [Back] << >> [Search]

Some later implementations of Windows may have the following buttons instead:

[Contents] [Search] [Back] [History] << >>

The Help text determines which of these buttons are enabled and which are grayed. The Help window has its own pop-up menus: **File**, **Edit**, **Bookmark**, and **Help**. In this section we'll discuss these menus and their menu items.

## File

This menu has four items. Two of them let you print the information, and one lets you close the Help window. The **Open...** menu item lets you load Help files from other Windows applications.

## Edit

The **Copy** menu item places the current text from the Help window into the clipboard. To add your own notes to the existing Help text, use the **Annotate...** item. A dialog box with an edit control appears. Enter your notes in the edit control.

As soon as you click **OK**, the dialog box disappears. A paper clip icon is inserted into the current text. This indicates that this topic contains user annotations, which can be displayed by selecting the **Annotate...** item.

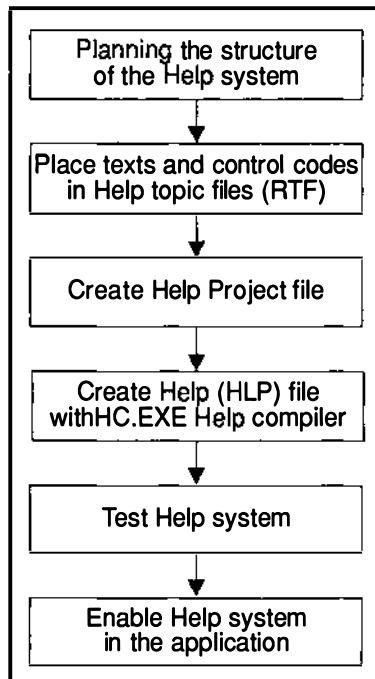
## Bookmark

It's also possible to add a bookmark to any topic and then use this bookmark to jump directly to the topic. Use any name you want for bookmarks. To display all existing bookmarks, select the **Bookmark** pop-up menu. When you click one of the names, the menu displays the appropriate topic.

## Help

The WINHELP.EXE help application also has its own Help file, which provides basic information on the Help topic. This text appears in the Help window when you select the **Using Help** menu item.

## Creating a Help system



In this section, we'll present a sample application that demonstrates how to create a Help system. This application is only capable of enabling the Help system and the individual items of the system. Passages from the

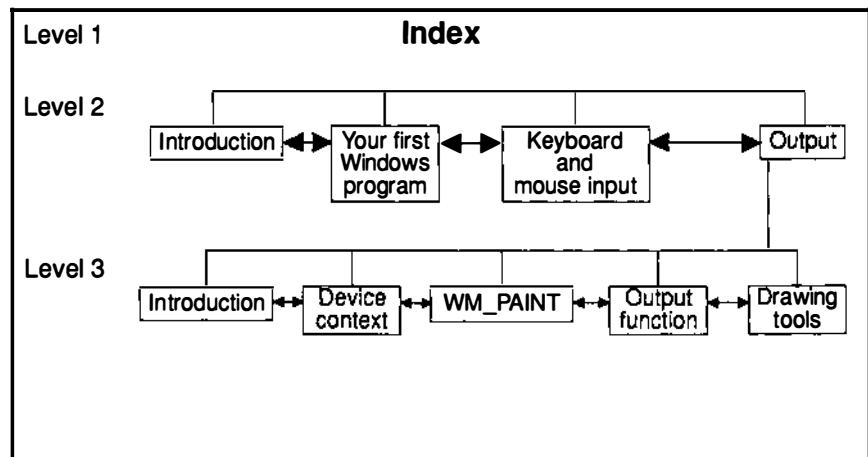
first four chapters of this book are included in this Help system. We'll explain each item in a separate section.

## Planning

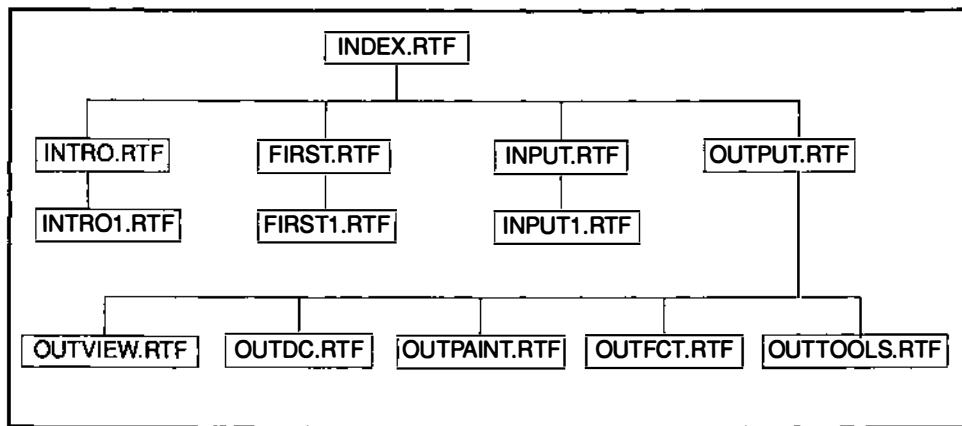
A Help system should provide complete information for the user. So, the first step in planning a Help system is to gather information. Often, much of this information is already available. For example, information can be gathered when the application is tested or can be taken from the user's manual, if one is available.

Once you have enough information, you must plan the structure of the Help system. The individual topics must be arranged hierarchically, with the Help index always at the top. Users should be able to move gradually from the top level to the bottom. The number of hierarchy levels depends on the number of topics and the size of the entire Help system.

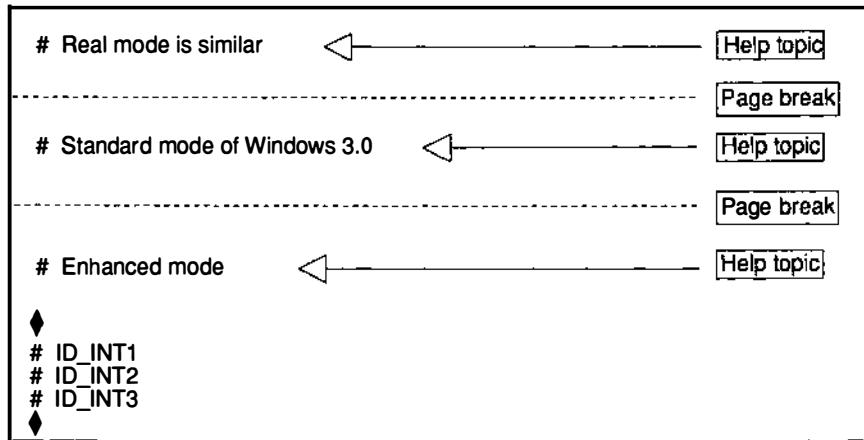
For example, in our Help system, the hierarchy looks as follows:



Also, you should be able to move directly from one topic to the next if the topics are on the same level. In the next step, we must plan the structure and the number of files containing topics.



Each Help topic describes a certain aspect of the application and can be in a separate file. If several topics are combined into a single file, separate them from each other with page breaks.



In our example, we used separate files for every topic, except for three definitions (see above).

All of the files must be saved as RTF (Rich Text Format). You'll need Microsoft Word for Windows, some implementation of Microsoft Word for PC (Version 5.0 or higher) or Macintosh (Version 4.0), or any other word processor that supports RTF format.

## Help topic files

When developing a Help system, most of your time will be spent creating Help topic files. First you must place the text in the proper file structure and then set control codes that determine how the user can move within the Help system. You can either insert these codes while editing the text or add them to the existing text later.

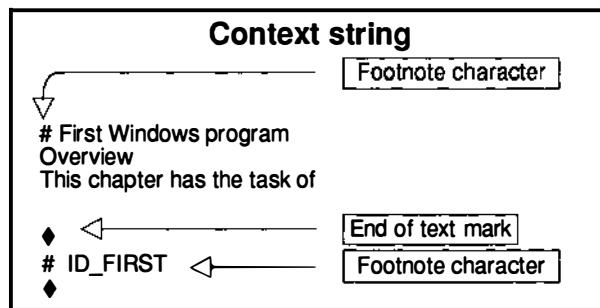
To create Help topic files, you need an editor that can save the files in RTF (Rich Text Format). Also, you must be able to work with footnotes, underline, and strikethrough character formats, since these are the attributes that constitute control codes. For example, Word 5.0, which we used to create the examples you'll find on the companion diskette for this book, fulfills these requirements.

There are seven different control codes, which you can create either from footnote characters or formatted text:

Control	Meaning
As footnote character:	
#	Refers to a context string
\$	Refers to the title of the topic
*	Refers to a switch that states whether the topic should be in the Help file
K	Refers to one or more keywords
+	Refers to list that specifies browse sequence
Character formatting:	
Strikethrough	Specifies a jump to a different topic
Underline	Specifies that there is a more precise definition of the underlined text

## Context string

Every topic in the Help system needs its own context string for identification. A valid context string can be up to 255 letters, numbers, and underscores (\_). Letters can be uppercase or lowercase. The strings are used for jumping between topics (see the Cross reference heading in this chapter). Only the index, which displays the topic at the top level, doesn't need a context string.



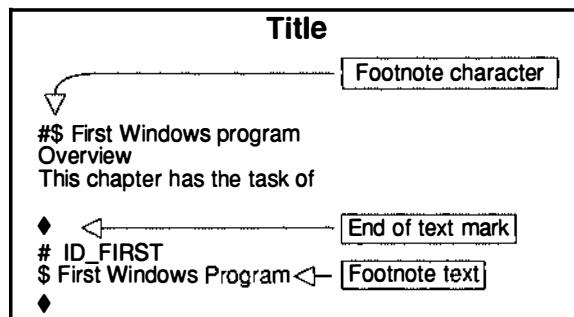
You assign a context string to a Help topic by marking the beginning of the topic with the # footnote character (see the above illustration). Then enter a space and the context string as footnote text. There can only be one space between the # character and the string.

## Title

Most Help topics have a title. In this instance we are referring to a title used in the Help system, instead of the title that's usually in the first line of the text.

For example, this title is presented to the user as a model when a bookmark is defined. It also appears in the "Go to" list in the Search dialog box when a search for a keyword is successful.

To define a title, enter the \$ footnote character at the beginning of the text for the appropriate topic. The title is then written as the topic's footnote text, which cannot be formatted.



Although the title can be up to 127 characters long, including spaces, we recommend that you limit your titles to 40 characters. This ensures that the title will be displayed in the "Go to" list.

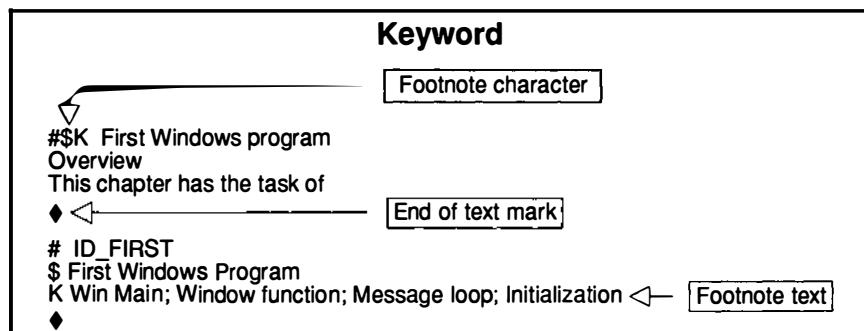
## Keyword

One of the best ways to use a Help system is to search for certain terms and then jump directly to the passage containing the term. Keywords make this process possible.

You can define as many keywords for a topic as you want. The keywords then appear in a list box when you select the **Search** button.

Again, the application doesn't distinguish between uppercase letters and lowercase letters. If several topics have the same keyword, the "Go to" list alphabetically displays the titles of all relevant topics.

Then the user can select the desired topic. The footnote character for the keywords is the uppercase K. Enter all of the keywords, separated by hyphens, as the footnote text for the topic. This text can also be several lines.

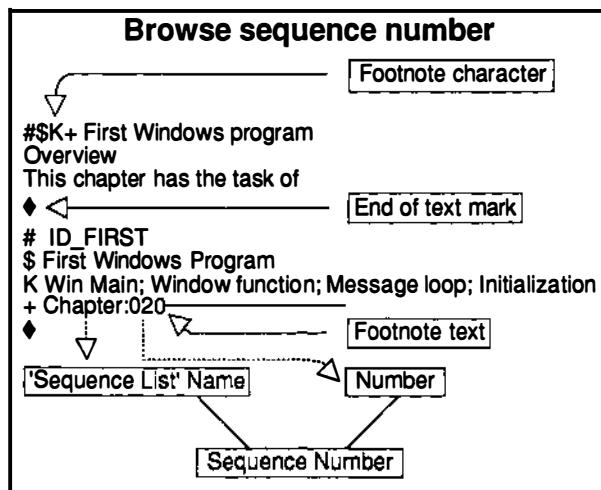


## Browse sequence number

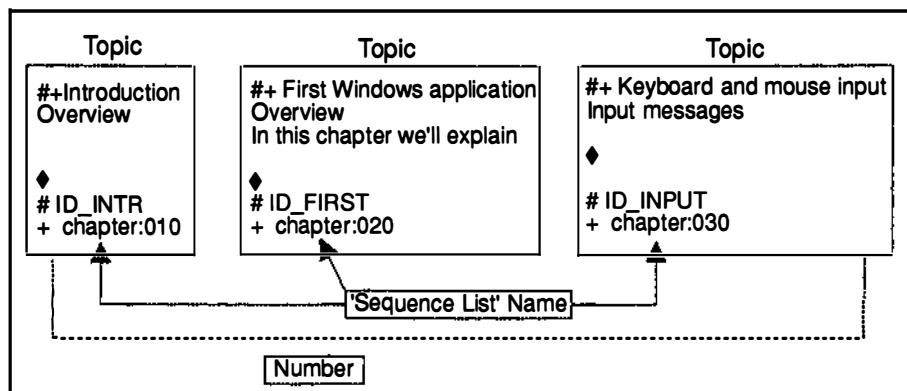
The **<<** and **>>** buttons enable the user to move among related topics.

The sequence in which the topics are presented is called the browse sequence. The creator of the Help system determines the browse sequence with sequence numbers.

The + footnote character is used for the browse sequence. The sequence number usually consists of a sequence name and a number, separated by a colon.



All sequence numbers with the same name belong to a browse sequence. If you don't specify a name, the topic is noted in a null list. The number determines the sequence within the browse sequence. Often numbers aren't assigned sequentially in order to leave room for extensions.



Since the three topics have the same sequence name, you can jump from one topic to the next. You can determine whether you have this option by clicking both **Search** buttons. As soon as you reach the beginning or

the end of the sequence list, the relevant button is automatically set to F-DISABLED status so that it can no longer be used.

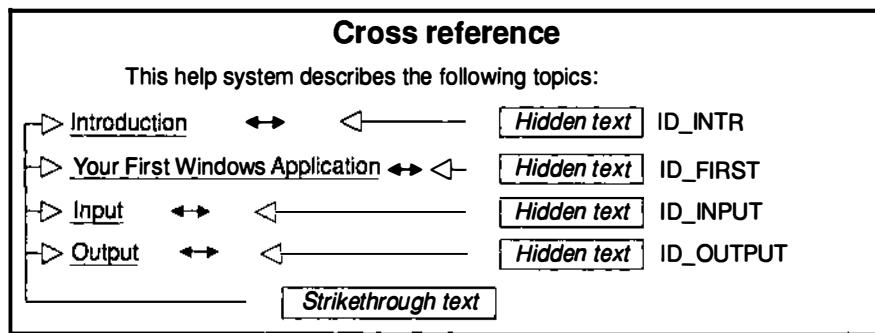
## Cross reference

Existing cross references enable you to go from the highest hierarchical level to the next until you reach the lowest level. Context strings of separate topics are needed to jump from one level to the next.

First you must determine a passage from where you want to jump. This text must be formatted in strikethrough character format.

Then, as hidden text, enter the context string of the topic to which you want to jump. The hidden text must be attached directly to the strikethrough text.

In Word 5.0, text formatted as strikethrough still appears normal in the display. In the following illustration we made the strikethrough characters visible so that you can see what they look like.



This illustration demonstrates how to jump by displaying the passage as underlined. The Help compiler converts the strikethrough character format to underline.

This help system describes the following topics:

Introduction

Your First Windows Application

Input

Output

## Definitions

Topic files often contain words or expressions that must be defined (e.g., foreign words).

To avoid cluttering the text with these definitions, the Help system enables the user to display the definitions only when needed. A dotted line appears under the expressions that contain definitions.

Introduction

Overview

Windows 3 was introduced in the spring of 1990. The Windows 3.0 graphic user interface is considered a user environment of DOS that is capable of multitasking.

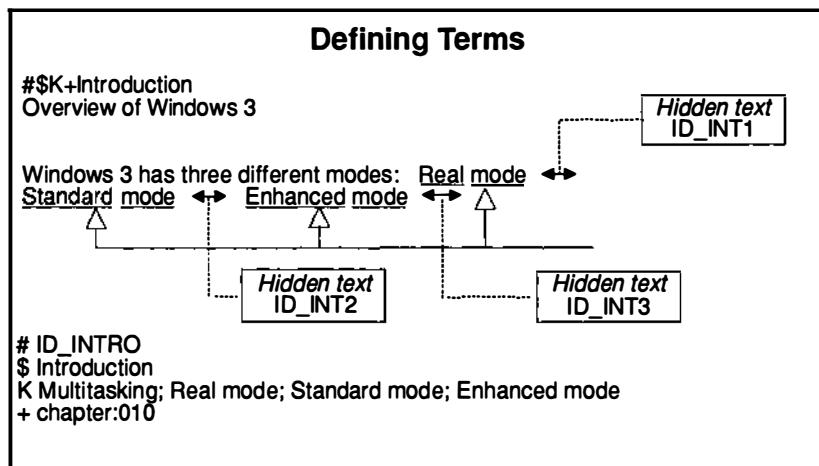
Windows Modes

Windows 3 has three different modes: Real mode, standard mode and enhanced mode.

When the user selects (with the mouse) text that is underlined in this way, a box appears with additional information. The additional text disappears when you release the mouse button.

According to the Help compiler, these additional definitions are independent topics. So they also require a context string. These definitions can either be combined into one separate file (the same file as the topic to which they refer) or separate files for each definition.

Expressions that have definitions appear in underlined character format. Then the context string of the topic, which provides the definition, is written as hidden text.



In Word 5.0 a text that has been formatted as underlined is still displayed as normal on the screen. So, in the above illustration we made these characters visible so that you can see what they look like.

## BUILDTAG

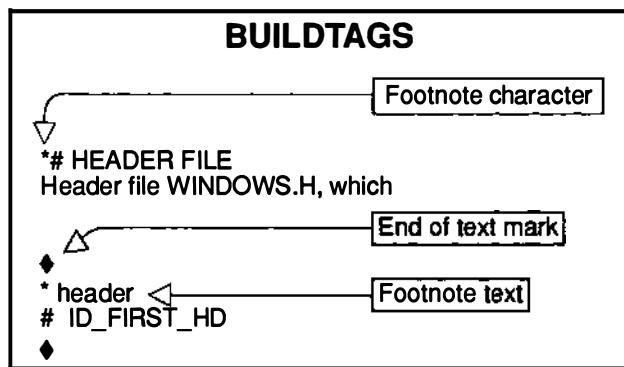
BUILDTAGS are switches with names that can be inserted into a topic. This is done so that the Help compiler can build the topic into the Help file.

This option is useful if you've created Help files for different versions of an application and one version has more attributes than the others.

The \* footnote character, which was described at the beginning of this topic, is used for BUILDTAGS. The Help compiler will work more efficiently if you insert this character in front of all the other footnote characters. Enter the name of the switch as the footnote text.

When you insert the build tag footnote, the switch is set to TRUE. In addition, all the switches in the help project file are specified in the [BuildTag] section. Now you can change the status of the switch in the file under the BUILD keyword in the [Options] section (refer to the next chapter).

Only topics with switches whose BUILDTAGS are set to TRUE are accepted into the Help file by the Help compiler.



All topics without switches are automatically written to the Help file. Therefore, switches are mainly used to exclude certain topics from the Help file. To exclude a topic, set the switch to FALSE. In our example, we excluded the header file topic by setting the switch to FALSE (see below).

## Help Project file

The Help project file contains all the information needed by the Help compiler to convert Help topic files into a WINHELP.EXE compatible binary help file.

This Help project file is a text file with an .HPJ extension. It consists of a maximum of five sections.

Section	Meaning
FILES	Lists all Help topic files needed
OPTIONS	Lists up to four different options
BUILDTAGS	Assigns valid build tags
MAP	Adds context strings with numbers
ALIAS	Assigns multiple topics to the same context string

Although the section order is usually arbitrary, the [Alias] section must precede the [Map] section in a few instances.

A semicolon must precede any comments that are added. When the Help compiler encounters a semicolon, it ignores any text following that character in the current line.

## [FILES]

Every Help project file needs this section. The [FILES] section lists the Help topic files the developer wants converted to a binary help file. The Help compiler searches the current directory and the path specified by the Root option for these files. If it cannot find a file, the compiler displays an error message. Here's an example of the HELP.HPJ file:

```
[FILES]
INDEX.RTF      ; Highest hierarchy level
INPUT.RTF       ; Chapter: Keyboard and Mouse Input
OUTPUT.RTF      ; Chapter: Output
FIRST.RTF       ; Chapter: Your First Windows Application
```

A large Help system can contain many files that interact with one another. Often the names of these files are placed into a header file for clarity. You can use an include statement to link this file to the [FILES] section. Here's a sample HELPINCL.H header file, and a HELP.HPJ file modified to accept data from the header file:

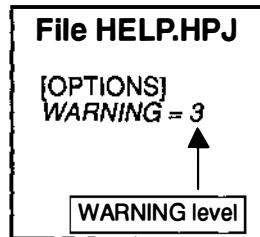
```
;   HELPINCL.H
INDEX.RTF      ; Highest hierarchy level
INPUT.RTF       ; Chapter: Keyboard and Mouse Input
OUTPUT.RTF      ; Chapter: Output
FIRST.RTF       ; Chapter: Your First Windows Application

;   HELP.HPJ
[FILES]
#include <HELPINCL.H>
```

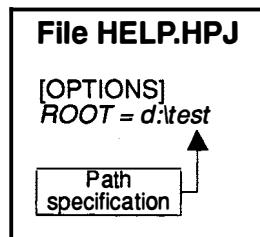
## [OPTIONS]

The [OPTIONS] section can contain up to four different options.

Use WARNING to specify the kinds of error messages the Help compiler outputs. There are three error message categories (1,2,3). If you set the error level to 1, only serious error messages are output. If you select level 3, all other error messages are displayed.

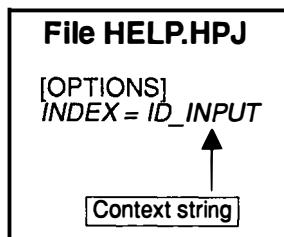


The Root option determines the root directory of the Help project. The Help compiler searches the root directory for the Help topic files. If you don't include a root directory, the compiler assumes the current directory.



If the Help project file contains only the [FILES] section, the Help compiler enters the topic of the first file as the highest hierarchy level.

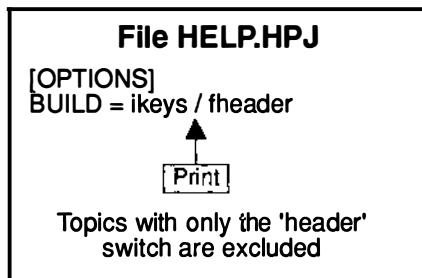
In this case, the index doesn't need a context string. You can also set the index with the Index option. Then you would enter the necessary context string of the desired topic.



To obtain the Index topic, press the Index button in the Help window.

If build tags have been built into the Help topic files, use the Build option to link the desired topics. Otherwise, this option isn't useful.

You can link all of the switches together by using the logical operators OR (!), AND (&), and NOT (~). When creating the Help file, the Help compiler only considers topics whose switches have been set to TRUE.



## [BUILDTAGS]

If you want to work with BUILDTAGS, the Help project file must include the [BUILDTAGS] section, as well as the Build option. The names of valid switches are listed in this section; a maximum of 30 switches are available. Here's the [BUILDTAGS] section as it appears in the HELP.HPJ file:

```
[BUILDTAGS]
ikey          ; Topic: Keyboard input
fheader       ; Topic: Header file
```

## [MAP]

If you want the Help system to support context-sensitive help, the [MAP] section must be included in the Help project file.

Context-sensitive Help provides help text for topics outside the client area. For example, the user could obtain information about the system menu by selecting the menu with the mouse. The user could also access information about certain key combinations.

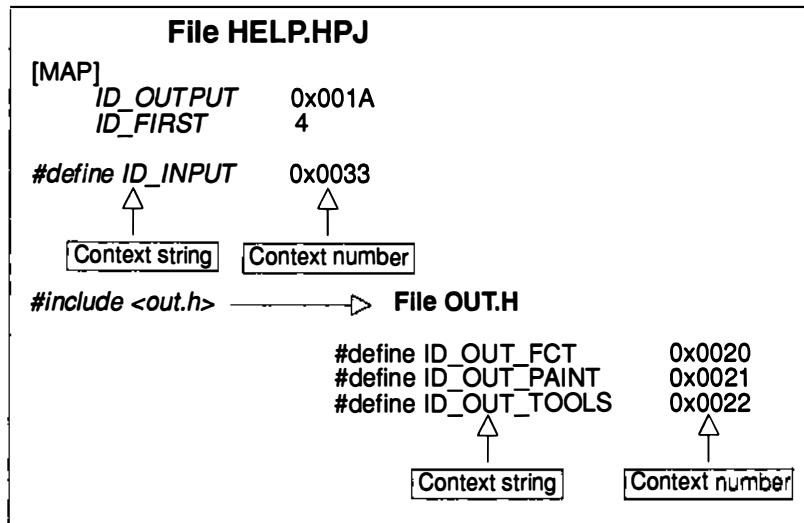
To call context-sensitive help, press **Shift** + **F1**. When you press this key combination, a question mark is added to the normal pointer (cursor), which indicates that you're in Help mode.

The programmer is responsible for programming such things as the cursor conversion and the messages that result by clicking outside of the client area.

Help text will only be provided for the appropriate topics. You can connect the appropriate topics by using context numbers.

In the [MAP] section, context strings are combined with context numbers. However, in order to display a certain topic, the context numbers must match the ones that the application sends at runtime.

Numbers can be written either in decimal or hexadecimal notation. You can also use a #define statement which, in turn, is in a separate header file that you can link by using an #include statement.



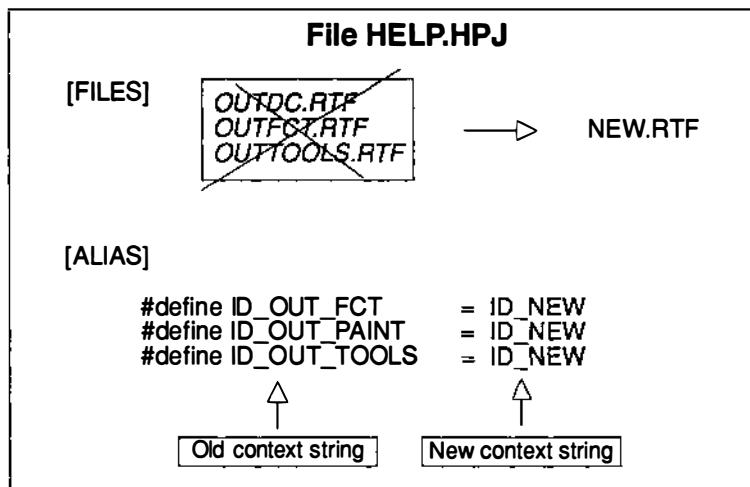
## [ALIAS]

The [ALIAS] section is required when the same context string is assigned to more than one topic. For example, this is useful when you want to replace two old topics with a new topic.

If you want to delete old topics but didn't use the [ALIAS] section, you must search all the other topics for invalid cross references. To avoid this, use [ALIAS]. The context string of the deleted topic is overwritten

by the string of the new topic. The new context string can also occur more than once.

In our example we deleted the three files with the topics Device Context, Output functions, and Drawing tools. The ID\_OUT\_DC, ID\_OUT\_FUNC, and ID\_OUT\_TOOLS context strings identified them. We replaced these files with the NEW.RTF file with the topic NEW, which has the ID\_NEW context string.



So the only file we must change is the Help project file, instead of the files of the other topics. In case you use alias names in the [MAP] section, the [ALIAS] section must precede the [MAP] section in the Help project file.

## The HC.EXE help compiler

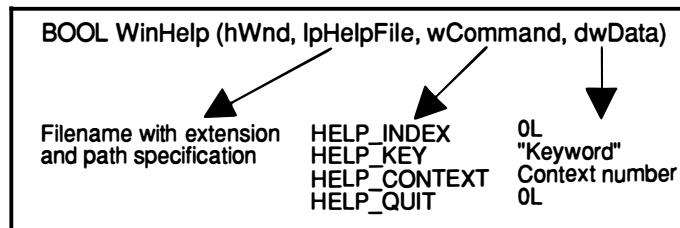
After the Help project file has been created, you can call the Help compiler using the filename as a parameter.

```
HC filename.hpj
```

The Help file that's created has the same name as the Help project file. The .HLP extension is added. Now you can load and read this new file using the WINHELP.EXE Help application.

# Programming the application

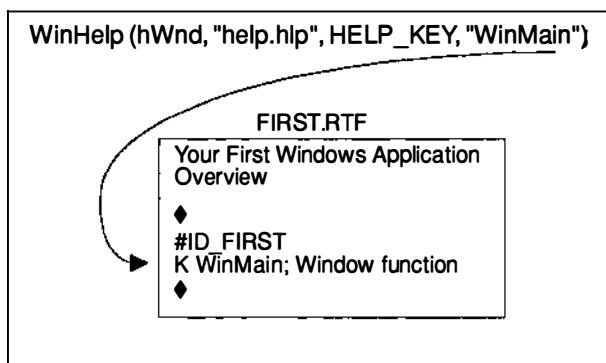
In order to activate the Help system from an application, you must use the WinHelp function.



Usually an application that provides help has a pop-up menu called **Help**. This menu contains a menu item called **Index**. When you click this item, the Index topic appears in the Help window. The WinHelp function is passed the value **HELP\_INDEX** as the third parameter. The dwData parameter is set to NULL.

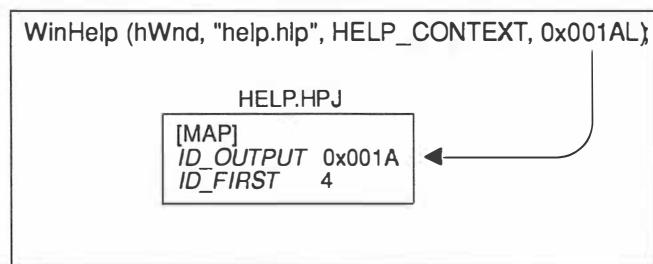
```
WinHelp( hWnd, "help.hlp", HELP_INDEX, 0L );
```

Frequently this pop-up menu contains other items that enable you to jump to other topics. In these instances, the wCommand parameter receives the value **HELP\_KEY**. The topic itself is specified by a keyword that's entered as such in the appropriate Help topic file.



If a context-sensitive Help has also been installed in the Help system, you can enable it with the **HELP\_CONTEXT** value. Then you need the

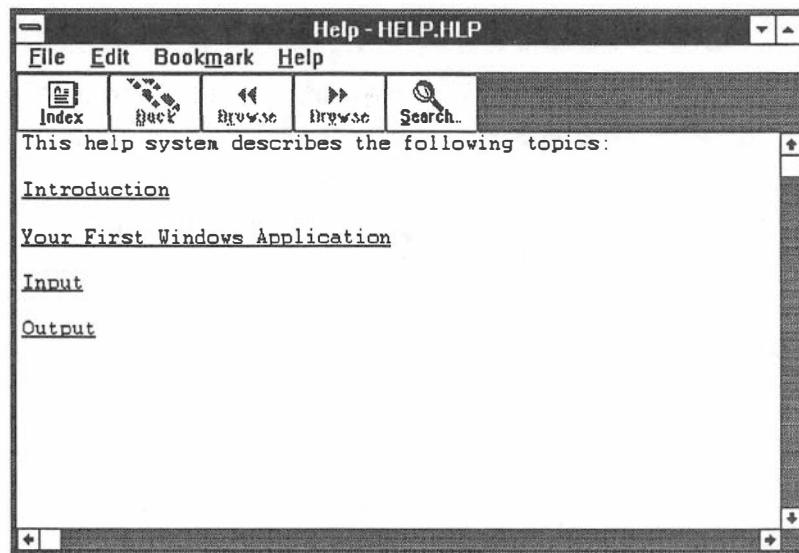
context number that was defined in the [MAP] section of the Help project file as your last parameter.



If you exit Windows, you must inform the Help application that you no longer need it. To do this, use the WinHelp function. The wCommand parameter is then HELP\_QUIT. If no other application is using Help, WINHELP.EXE is closed.

## Example of the Help system

We've already demonstrated parts of this application and its Help system in the preceding chapters. The following illustrations show this application at work under Windows 3.0 (your screens may look different):



The application contains a **Help** pop-up menu. You can use four menu items to get information on the following topics: **Index**, **Your 1st Application**, **Input**, and **Output**.

## Source code: HELP.C

```
/** HELP.C *****/
/** Help system demonstration. Uses HELP.HLP file compiled using HC.EXE.  ***/
/** *****/

#include "windows.h"                                // Include windows.h header file
#include "help.h"                                    // Include help.h header file

BOOL HelpInit(HANDLE);
long FAR PASCAL HelpWndProc(HWND, unsigned, WORD, LONG);

/** WinMain (main function for every Windows application) *****/

int PASCAL WinMain(hInstance, hPrevInstance, lpCmdLine, CmdShow)
HANDLE hInstance;                                     // Current instance
HANDLE hPrevInstance;                                // Previous instance
LPSTR lpCmdLine;                                    // Long ptr to string after name
int CmdShow;                                         // Specifies app. window
{
    MSG msg;                                         // Message variable
    HWND hWnd;                                        // Window handle
    HANDLE hAccel;                                    // Accelerator handle

    if (!hPrevInstance)                               // Initialize first instance
    {
        if (!HelpInit(hInstance))                  // If initialization fails
            return (FALSE);                         // return FALSE
    }
}

/** Specify appearance of application's main window *****/

hWnd = CreateWindow("Help",                            // Window class name
                    "Help system demo",           // Window caption
                    WS_OVERLAPPEDWINDOW,         // Overlapped window
                    CW_USEDEFAULT,              // Default upper-left x pos.
                    0,                           // Upper-left y pos.
                    CW_USEDEFAULT,              // Default initial x size
                    0,                           // y size
                    NULL,                        // No parent window
```

```
        NULL,                                // Window menu used
        hInstance,                           // Application instance
        NULL);                             // No creation parameters

ShowWindow(hWnd, CmdShow);           // Make window visible
UpdateWindow(hWnd);                // Update window

hAccel = LoadAccelerators(hInstance, "Help"); // Load kbd accelerators

while (GetMessage(&msg, NULL, NULL, NULL))    // Message reading
{
    if (!TranslateAccelerator(hWnd, hAccel, &msg))
    {
        TranslateMessage(&msg);          // Message translation
        DispatchMessage(&msg);          // Send message to Windows
    }
}
return (msg.wParam);                // Return wParam of last message
}

BOOL HelpInit(hInstance)            // Initialize instance handle
HANDLE hInstance;                  // Instance handle
{

/** Specify window class *****/
WNDCLASS wcHelpClass;              // Main window class

wcHelpClass.style      = NULL;       // Horiz. & vert. redraw
wcHelpClass.lpfnWndProc = HelpWndProc; // Window function
wcHelpClass.cbClsExtra = 0;          // No extra bytes
wcHelpClass.cbWndExtra = 0;          // No extra bytes
wcHelpClass.hInstance  = hInstance;   // Instance
wcHelpClass.hIcon      = LoadIcon(NULL, IDI_APPLICATION); // Icon
wcHelpClass.hCursor    = LoadCursor(NULL, IDC_ARROW);     // Cursor
wcHelpClass.hbrBackground = GetStockObject(WHITE_BRUSH); // Background
wcHelpClass.lpszMenuName = "HelpMenu";                   // Help menu
wcHelpClass.lpszClassName = "Help";                      // Window class

if (!RegisterClass(&wcHelpClass)) // Register window class
    return FALSE;
return (TRUE);
}
```

```
/** HelpWndProc ****
/* Main window function: All messages are sent to this window      */
/*****
long FAR PASCAL HelpWndProc(hWnd, message, wParam, lParam)
HWND      hWnd;                                // Window handle
unsigned message;                            // Message handle
WORD      wParam;
LONG      lParam;
{
    char   chFileName[10] = "help.hlp"; // Help file's name

    switch (message)                      // Process messages
    {
        case WM_COMMAND:
            switch (wParam)
            {
                case IDM_HELP_INDEX: // Get Index
                    WinHelp(hWnd, chFileName, HELP_INDEX, 0L);
                    break;

                case IDM_HELP_FIRST: // Get First app. help

WinHelp(hWnd, chFileName, HELP_KEY, (DWORD) (LPSTR) "WinMain");
                    break;

                case IDM_HELP_INPUT: // Get Input help

WinHelp(hWnd, chFileName, HELP_CONTEXT, 0x0001L);
                    break;

                case IDM_HELP_OUTPUT: // Get output help

WinHelp(hWnd, chFileName, HELP_CONTEXT, 0x000aL);
                    break;

                default:           // Send other msgs here
                    return (DefWindowProc(hWnd, message, wParam,
lParam));
            }
            break;

        case WM_DESTROY:          // Destroy window
            WinHelp(hWnd, chFileName, HELP_QUIT, 0L);
            PostQuitMessage(0);
            break;
    }
}
```

```
    default:
        return (DefWindowProc(hWnd, message, wParam, lParam));
}

return (0L);
}
```

## Module definition file: HELP.DEF

```
NAME      Help

DESCRIPTION 'Help example'

EXETYPE   WINDOWS

STUB      'WINSTUB.EXE'

CODE      PRELOAD MOVEABLE

DATA      PRELOAD MOVEABLE MULTIPLE

HEAPSIZE  4096
STACKSIZE 4096

EXPORTS   HelpWndProc
```

## Resource script: HELP.RC

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

HelpMenu      MENU
BEGIN
    POPUP    "&Help"
    BEGIN
        MENUITEM "&Index",           IDM_HELP_INDEX
        MENUITEM "Your 1st &Application", IDM_HELP_FIRST
        MENUITEM "I&nput",           IDM_HELP_INPUT
        MENUITEM "&Output",          IDM_HELP_OUTPUT
    END
END

Help ACCELERATORS
BEGIN
    VK_F1, IDM_HELP_INDEX, VIRTKEY
END
```

## Header file: HELP.H

```
#define IDM_HELP_INDEX 101
#define IDM_HELP_FIRST 102
#define IDM_HELP_INPUT 103
#define IDM_HELP_OUTPUT 104
```

The **F1** function key was defined as an accelerator in the .RC file so that you could also press this key to activate the Help system. Also, the two necessary functions for accelerator keys were inserted into the program text. The wCommand parameter of the WinHelp function is set to HELP\_CONTEXT in two cases, as well as when help is not handled as context-sensitive.

However, along with the HELP\_KEY value you can also use the HELP\_CONTEXT value to jump directly to a topic as long as there is a context number defined for this topic.

There are a total of thirteen Help topic files that were created with Word 5.0 and then converted to RTF format. All the topics, except the Index, have a title; some also have keywords.

We already documented the relationships the individual topics have to one another in the section on planning. In Word 5.0, a text formatted as underlined or strikethrough is displayed in normal text. To illustrate the character formats, the underline or strikethrough are visible in the following figures.

**INDEX.RTF:**

This Help system describes the following topics:

**IntroductionID\_INTR**

**Your First Windows ApplicationID\_FIRST**

**InputID\_INPUT**

**OutputID\_INPUT**



**FIRST.RTF:**

#\$K+Your First Windows Application  
Overview

In this chapter we'll explain the important

HEADER fileID\_FRSTHD



# ID\_FIRST

\$ Your First Windows Application

K WinMain; Window function; Message loop; Initialization

+ chapter:020



**FIRST1.RTF:**

\*#Header file

The header file WINDOWS.H, found in



\*fheader

# ID\_FRSTHD



INTRO.RTF:

#\$K+Introduction  
Overview of Windows 3  
Windows 3 was introduced in the spring of 1990

Windows Modes  
Windows 3.0 has three different modes: Real modeID\_INT1,  
standard modeID\_INT2 and enhanced modeID\_INT3

◆  
# ID\_INTR  
\$ Introduction  
K Multitasking; Real mode; Standard mode; Enhanced mode  
+ chapter:010

◆

INTRO1.RTF:

# Real mode is very similar to the segmented addressing used

-----  
# Standard mode exceeds the 1 Meg limit set by MS-DOS and

-----  
# In the enhanced mode, which can be used only on the 80386 or

◆  
# ID\_INT1  
# ID\_INT2  
# ID\_INT3

◆

INPUT.RTF:

#\$K+ Keyboard & mouse input  
Input messages  
In this chapter we'll describe the messages

Keyboard input ID\_IN\_KBD

◆

# ID\_FIRST\_HD  
\$ Keyboard and mouse input  
K Queue; Mouse; Timer; Keyboard  
+ chapter:030

◆

**INPUT1.RTF:**

\*# Keyboard input  
Keyboard messages  
As soon as the user presses a key

◆  
\*ikeys  
# ID\_IN\_KBD  
◆

**OUTPUT.RTF:**

#\$+Output:  
OverviewID-OUT\_VIEW  
Device\_contextID\_OUT\_DC  
WM\_PAINT\_messageID\_OUT\_PAINT  
Output\_functionsID\_OUT\_FCT  
Drawing\_toolsID\_OUT\_TOOLS

◆  
# ID\_OUTPUT  
\$ Output  
+ chapter:040  
◆

**OUTDC.RTF:**

#\$+Device context  
Definition  
You must have a device context in order

◆  
# ID\_OUT\_DC  
\$ Device Context  
+ out:020  
◆

**OUTPAINT.RTF:**

#\$+KThe WM\_PAINT message  
In Windows applications you should

◆  
# ID\_OUT\_PAINT  
\$ WM\_PAINT  
+ out:030  
K WM\_PAINT; Queue  
◆

**OUTVIEW.RTF:**

#\$+Overview  
In this chapter we'll examine

◆  
# ID\_OUT\_VIEW  
\$ Overview  
+ out:010  
◆

**OUTFCT.RTF:**

#\$+Output functions  
K The simpler output functions are divided into the basic text and

◆  
# ID\_OUT\_FCT  
\$ Output functions  
+ out:040  
K Text; Graphics; Device context; Mapping mode; Pixel; TextOut;  
DrawText; DC  
◆

**OUTTOOLS.RTF:**

```
#$+Drawing tools  
Basics  
It's possible to change some GDI  
  
◆  
# ID_OUT_TOOLS  
$ Tools  
+ out:050  
◆
```

The Help project file consists of the [FILES], [MAP], [BUILDTAGS], and [OPTIONS] sections. The two [BUILDTAGS] defined in this file are used in the INPUT1.RTF and FIRST1.RTF files. Since the header switch in the [OPTIONS] section is set to FALSE, the topic that this switch contains is included in the Help file.

## **Help project file: HELP.HPJ**

```
[FILES]  
INDEX.RTF  
OUTVIEW.RTF  
OUTDC.RTF  
OUTPAINT.RTF  
INPUT.RTF  
OUTPUT.RTF  
OUTTOOLS.RTF  
OUTFCT.RTF  
INTRO.RTF  
INTRO1.RTF  
INPUT1.RTF  
FIRST.RTF  
FIRST1.RTF  
  
[MAP]  
#define ID_INPUT    0x0001  
     ID_OUTPUT   10  
  
[BUILDTAGS]  
ikeys      ; Keyboard input  
fheader    ; Header files  
  
[OPTIONS]  
BUILD=ikeys | fheader
```

# **The Multiple Document Interface**

## **General information**

The Multiple Document Interface (MDI) provides a standardized interface for presenting and processing several documents within a single application. An MDI application consists of a main window, in which the user can open any number of documents that he/she wants to change. Each document appears in its own window, which is a child window containing a system menu, etc.

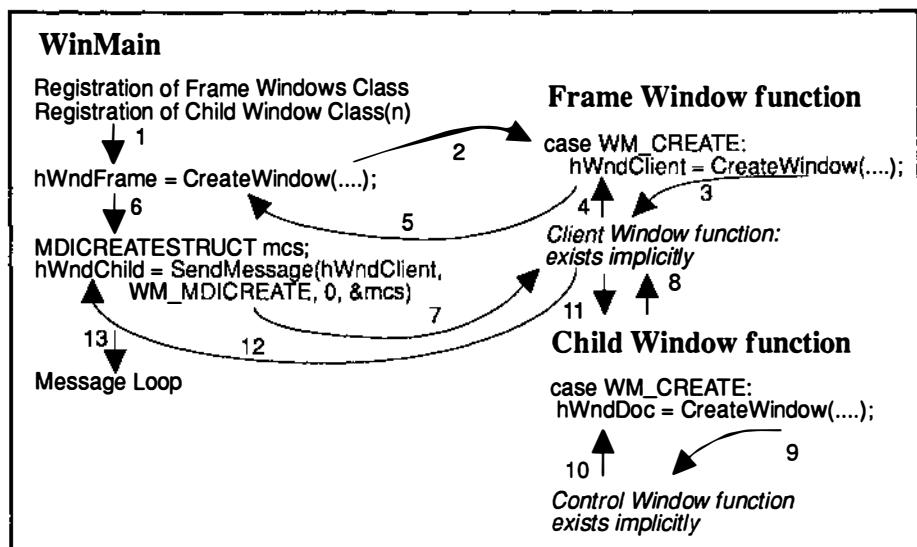
The child windows are coordinated with the help of messages that are sent from the various hierarchy levels (frame window <-> client window <-> child window). The client window, which takes the place of the client area in the frame window and is controlled by Windows, takes over many of the management tasks.

## **Structure of an MDI application**

The following illustration shows the basic structure of an MDI application. The individual points, such as the registration of the classes for the child window or the MDICREATESTRUCT structure, are explained in the following sections.

If the MDI child window is supposed to fulfill the same tasks as the controls of the "edit" or "list box" classes, then frequently a control window with the appropriate class is placed over the MDI child window.

In the following illustration, only the first child window is created. The others are normally generated based on results taken from the frame window's window function. The client window can also be created in the WinMain function.



## Additions and changes

In earlier versions of Windows (prior to Windows 3.0), creating MDI applications was extremely difficult because the programmer had to handle all the details. Windows 3 manages the MDI window itself, but linking MDI functions requires more time and effort when compared to normal applications.

## Message Loop

The child windows also have accelerators in the system menu. However, instead of using **Alt** to obtain them, you use the **Ctrl** key. For example, you could use the **Ctrl** + **F4** accelerator to close a child window. When you're ready to edit the next document, use the **Ctrl** + **F6** key combination to bring it to the foreground.

For these accelerator keys to be effective, you must use the TranslateMDISysAccel function to expand the message loop. This function checks whether the accelerator of a child window was pressed. If the function finds WM\_KEYDOWN messages with the appropriate keys, it translates them into WM\_SYSCOMMAND messages and then sends the messages to the active child window.

Then the function returns the value of TRUE. Otherwise, the return value is FALSE. The TranslateAccelerator function can check whether the appropriate accelerator key is present in the user-defined accelerator table for the frame window.

```
while( GetMessage( &msg, NULL, 0, 0 ) )
{
    if( !TranslateMDISysAccel( hWndMDIClient, &msg ) &&
        !TranslateAccelerator( hWndFrame, hAccel, &msg ) )
    {
        TranslateMessage( &msg );
        DispatchMessage( &msg );
    }
}
```

## Frame window

The main window is named in an MDI application frame window and almost always has a menu. There aren't any special characteristics to consider when you register its class. When the window is created in the WinMain routine, you should pass the WS\_CLIPCHILDREN style parameter to the CreateWindow function. Otherwise, the client window could be overwritten during output.

All messages that aren't processed in the window function of the frame window aren't sent to the DefWindowProc default routine window. Instead, they are sent to the DefFrameProc routine, which has five transfer parameters instead of four.

```
LONG DefFrameProc(hWnd, hWndMDIClient, wMsg, wParam, lParam)
```

*MDI Frame-Window*

*MDI Client-Window*

Because of the Multiple Document Interface, this function must handle additional tasks for certain messages. For example, if the function receives a WM\_SIZE message, it adapts the size of the client window to the new dimensions of the client area.

The function passes a WM\_SETFOCUS message to the window function of the client window, which then assigns the focus to the active child

window. This is why these two messages should always be passed to the DefFrameProc routine, regardless of whether they have also been processed.

All WM\_COMMAND messages that result from selecting a child window must also be passed to the default routine so that the user can enable the child window from the menu. The routine also sends many unprocessed messages to the normal DefWindowProc function.

The frame window often has a menu where the title of the MDI child window is also displayed as a menu item (see below). During application runtime this menu can be exchanged by sending the WM\_MDISETMENU message to the MDI client window. The lParam parameter contains the two handles of the menu and the pop-up menu, under which the title of the child window should appear.

## MDI client window

From the user's point of view, the client window provides the background of the main window where the child windows appear. For the programmer, this is a window with a predefined class named "mdiclient". Therefore, it also has a predefined window function.

A client window cannot be created until after the frame window because the handle of the frame window is the hWndParent parameter of the CreateWindow function. Therefore, the client window is either created in the WinMain function immediately after the frame window is created or in the WM\_CREATE message in the window function of the frame window.

```
Class
|
CreateWindow( "mdiclient", NULL,
              WS_CHILD | WS_CLIPCHILDREN | WS_HSCROLL | WS_VSCROLL,
              0, 0, 0
              hWndFrame, NULL, hInstance, (LPSTR)&ccs);

```

Frame-Window = Parent WindowVariable of CLIENTCREATESTRUCT structure

The client window receives scroll bars from the two style parameters, WS\_HSCROLL and WS\_VSCROLL. However, the scroll bars are only visible when the frame window is too small to display the complete child

windows. Without the WS\_CLIPCHILDREN style, the child windows would be painted over.

## **CLIENTCREATESTRUCT**

The last parameter, lParam, points to a variable of the CLIENTCREATESTRUCT structure. This structure consists of two fields:

**hWindowMenu** Handle of the menu to which the title of the MDI child window is attached

**idFirstChild** ID value of the first MDI child window

The handle of the pop-up menu, to which the title of the existing child windows should be added as menu items, is given to the first field. You could use, for example, the GetSubMenu function to obtain the menu handles.

Now the user can place the document to be edited in the foreground by selecting the desired title from the pop-up menu.

The ID value of the first created MDI child window is given in the second field. This value is incremented for each of the following child windows. When a child window is closed, Windows assigns new values to the windows so that there aren't any gaps within the value range.

Since the titles of the child windows are treated like normal menu items, when you click them they produce a WM\_COMMAND message that is sent to the window function of the frame window.

The supplied ID value of the menu item is identical to the ID value of the MDI child window. So, when you assign the first window IDs you should ensure that the ID values don't conflict with the other menu IDs.

## **MDI child window**

The child windows in an MDI application are very similar to a normal main window. They have a system menu, a minimize and a maximize

box, and can be resized. The only difference is that you can move them from the frame window.

An MDI application can have one or more classes for the MDI child windows. This number depends on the application's task.

Since each child window within the frame window can be displayed as an icon, you should pass an icon handle to the hIcon field in the class structure. You can also reserve a special place for data that refers to a certain window by using the cbWndExtra field.

Use the GetWindowWord and GetWindowLong functions to obtain this data from the reserved area. To save the data there, use the SetWindowWord and SetWindowLong functions. Besides data, these functions can also be used to prompt certain fields from the window and class structure. The two functions with the Long syllable work with long integer data.

Therefore, they use an offset that must be increased by four bytes each time. The other two functions save and load WORD data. These functions use an offset that is two bytes in size. The offset always starts at Null with its own data.

```
LONG GetWindowLong( hWnd, nIndex)
WORD GetWindowWord( hWnd, nIndex)
Offset from 0 to n
LONG SetWindowLong( hWnd, nIndex, dwNewLong)
WORD SetWindowWord( hWnd, nIndex, wNewWord)
```

Special MDI messages control the child windows in an MDI application. Usually, the client window receives an MDI message on the basis of which child window is created, closed, enabled etc. For example, the messages can result when a menu item is selected from the menu of the frame window.

**Frame window function**

case WM\_COMMAND:

▪

case ID\_END:

    SendMessage(hWndMDIClient, WM\_MDIDESTROY, hWndChild, 0L);  
    break;*Client window function:***Client window function:**

case WM\_DESTROY:

    return DefMDIChildProc(hWndChild, msg, wParam, lParam);  
    break;

Frequently the messages only affect the window that is currently active. However, there are three MDI messages that affect all MDI child windows because they rearrange the windows.

WM\_MDICASCADE

WM\_MDIICONARRANGE

MDI\_TILE

The two parameters wParam and lParam are always set to null. The WM\_MDICASCADE message places windows on top of one another, starting at the upper-left corner and proceeding to the lower-right corner.

However, if you send the WM\_TILE message to the client area, the child windows are arranged next to each other. The WM\_MDIICONARRANGE message rearranges all icons representing the MDI child windows in the client window.

The CreateWindow function doesn't create an MDI child window. Instead the WM\_MDICREATE message is sent to the client window. The lParam parameter points to a variable of the MDICREATESTRUCT structure.

## MDICREATESTRUCT

szClass	Pointer to the registered class name of the MDI child window
szTitle	Pointer to the title of the MDI child window
hOwner	Instance handle
x	x position of the upper-left corner of the MDI child window
y	y position of the upper-left corner of the MDI child window
cx	Initial width of the MDI child window
cy	Initial height of the MDI child window
style	Additional styles for the MDI child window

The fields in this structure are very similar to the parameters of the CreateWindow function. Pointers to the previously registered class and a window title are needed. The hOwner field is given the current instance handle.

The initial coordinates and size are usually filled with the CW\_USEDEFAULT value, so that Windows can initially display the child windows as cascaded. The last field can be a combination of the four style parameters specified below, but you cannot use WS\_MAXIMIZE and WS\_MINIMIZE simultaneously because they each specify a special initial status of the window.

## Implicit MDI child window styles

WS\_CHILD  
WS\_CLIPSIBLINGS  
WS\_CLIPCHILDREN  
WS\_SYSMENU  
WS\_CAPTION

WS\_THICKFRAME  
WS\_MAXIMIZEBOX  
WS\_MINIMIZEBOX

## Optional MDI child window styles

WS\_MAXIMIZE  
WS\_MINIMIZE  
WS\_HSCROLL  
WS\_VSCROLL

Style parameters, such as WS\_CHILD and WS\_SYSMENU, which are both necessary in the CreateWindow function, are automatically used when a child window is created with the WM\_MDICREATE message.

You can increase the client area of an existing child window to the size of the entire client window at any time by sending WM\_MDIMAXIMIZE message to the client window. Windows then places the system menu of the child window to the left in the menu bar of the frame window and writes the name of the child window in the title bar of the frame window. Sending the WM\_MDIRESTORE message returns the child window to its original status.

The wParam parameter contains the handle of the child window. The WM\_MDINEXT message sets the next child window in the window list to active and places the window that was active behind all other child windows.

WM\_MDIMAXIMIZE  
WM\_MDIRESTORE  
WM\_MDINEXT

Since Windows manages the process of an MDI application, it uses these messages internally more than the application itself.

The WM\_MDIACTIVE message can go to the client window or the child window. The client window receives it if another child window should be enabled. For example, this occurs when the user presses **Ctrl** + **F6** to get to the next document.

After enabling the child window, the client window sends the message to the window that becomes inactive and to the new active child window. The lParam parameter contains both window handles. The wParam parameter is set to TRUE when the child window is set to active. When the message goes to the client window, the wParam parameter contains the handle of the child window that should be activated.

## **WM\_MDIACTIVATE**

to the client window:

```
wParam = Handle of child window to be activated  
lParam = NULL
```

to a child window:

```
wParam = TRUE or FALSE  
lParam = HIWORD( hWndNewActive )  
        LOWORD( hWndOldActive )
```

In the window function of the child window, either the handle of the active child window can be saved if it receives the WM\_MDIACTIVATE message or the handle can be determined with the WM\_MDIGETACTIVE message, which is sent to the client window.

```
Child window function: ( hWnd, msg, wParam, lParam )  
    case WM_MDIACTIVATE:  
        if( wParam )  
            hWndActive = hWnd;  
        break;  
  
        or  
  
    hWndActive = LOWORD( SendMessage( hWndClient, WM_MDIGETACTIVE, 0, 0L ) );
```

For example, if you set the frame window to active status by clicking the mouse, then the MDI child window that was last enabled with the WM\_MDIACTIVATE message doesn't receive an MDI message. Instead, it receives the WM\_MDIACTIVATE message only.

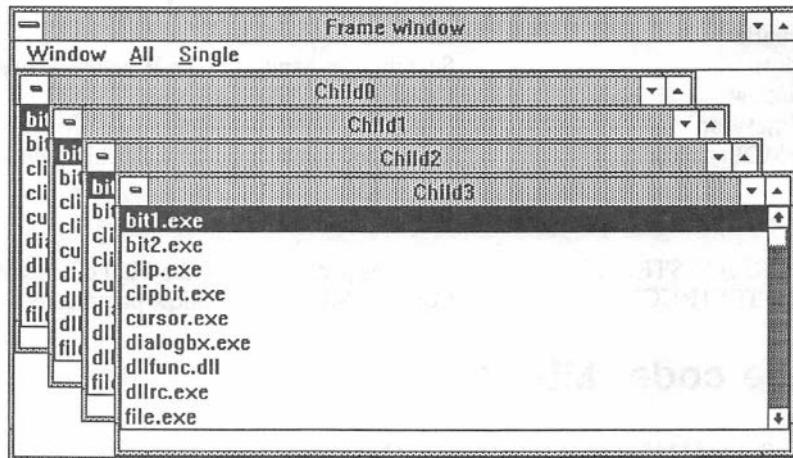
There is also a special predefined window function for the child windows that is called DefMDIChildProc and has four parameters.

```
LONG DefMDIChildProc(hWnd, wMsg, wParam, lParam)
```

MDI child window

Certain messages, such as WM\_SIZE, WM\_MOVE, WM\_CLOSE, etc. must always be passed to this routine so that the MDI application functions properly. As an example, the WM\_CLOSE message causes the DefMDIChildProc function to send the WM\_MDIDESTROY message to the client window. Programmers should also pass this MDI message to the client window when they want to close an MDI child window. Otherwise, it could disturb the MDI management.

## MDI example



This following application is a short example of MDI that demonstrates the fundamental process. The MDI child windows have their own child windows, which are from the predefined class Listbox and completely fill up the MDI child windows. All files of the current directory are displayed in the list boxes. The frame window has a menu that you can use to change the status of the MDI child windows. The menu allows you to create, close, and rearrange the child windows. Although the entire menu can be replaced by another, the titles of the existing child windows will be used as menu items.

<b>New messages</b>	<b>Brief description</b>
LB_GETCURSEL	Supplies index of items selected from a list box
LB_SETCURSEL	Selects a string in a list box
WM_MDIACTIVATE	Activates an MDI child window
WM_MDICASCADE	Arranges MDI child windows in cascade format
WM_MDICREATE	Creates an MDI child window
WM_MDIDESTROY	Closes an MDI child window
WM_MDIGETACTIVE	Supplies the handle of the active child window
WM_MDIICONARRANGE	Arranges the icons of all MDI child windows
WM_MDIMAXIMIZE	Sets an MDI child window to maximum size
WM_MDIEXT	Activates the next MDI child window
WM_MDIRESTORE	Restores an MDI child window to its previous size
WM_MDISETMENU	Changes the menu of the frame window
WM_MDITILE	Arranges all MDI child windows next to each other
<b>New functions</b>	<b>Brief description</b>
DlgDirList	Fills list box with specified directory
DrawMenuBar	Redraws the menu bar
GetWindow	Searches for handles in the Window Manager List
MoveWindow	Causes a WM_SIZE message
SetWindowWord	Changes a Window attribute
TranslateMDISysAccel	Translates accelerators of MDI child windows
<b>New structures</b>	<b>Brief description</b>
CLIENTCREATESTRUCT	Contains parameters for the MDI client window
MDICREATESTRUCT	Contains MDI child window parameters

## **Source code: MDI.C**

```
/** MDI.C ****
/** Demonstrates MDI (Multiple Document Interface) access by generating    */
/** child windows, frame windows and list boxes                         */
/** ****

#include "windows.h"                                // Include windows.h header file
#include "mdi.h"                                     // Include mdi.h header file

HWND hwndMDIClient = NULL;                          // Window handle to MDI client
                                                    // window

BOOL MDIInit(HANDLE);
LONG FAR PASCAL FrameWndProc(HWND, unsigned, WORD, LONG);
LONG FAR PASCAL MDIChildWndProc(HWND, unsigned, WORD, LONG);
```

```
/** WinMain (main function for every Windows application *****/
int PASCAL WinMain(hInstance, hPrevInstance, lpszCmdLine, CmdShow)
HANDLE hInstance;                                // Current instance
HANDLE hPrevInstance;                            // Previous instance
LPSTR lpszCmdLine;                             // Ptr to string after prg. name
int CmdShow;                                    // Specify app. window appearance
{
    MSG     msg;                                // Message variable
    HANDLE hAccel;                            // Accelerator handle
    HWND    hwndChild, hwndFrame = NULL; // Window handle
    MDICREATESTRUCT mcs;                     // Create structure variable

    if (!hPrevInstance)                      // Initialize first instance
    {
        if (!MDIInit(hInstance))           // If initialization fails
            return FALSE;                // return FALSE
    }

/** Specify appearance of application's main window *****/
    hwndFrame = CreateWindow ("Frame",                  // Window class name
                           "Frame window",          // Window caption
                           WS_OVERLAPPEDWINDOW | // Overlapped window style
                           WS_CLIPCHILDREN,        // child window style
                           CW_USEDEFAULT,          // Default upper-left x pos.
                           0,                      // Upper-left y pos.
                           CW_USEDEFAULT,          // Default initial x size
                           0,                      // y size
                           NULL,                  // No parent window
                           NULL,                  // Window menu used
                           hInstance,              // Application instance
                           NULL);                 // No creation parameters

    if (!hwndFrame)                            // If frame window fails
        return FALSE;                          // return FALSE

    if (!hwndMDIClient)                      // If MDI client window fails
        return FALSE;                          // return FALSE

    hAccel = LoadAccelerators (hInstance, "MDIMenu"); // Get accelerators
                                                    // for MDI
    ShowWindow (hwndFrame, CmdShow);         // Make window visible
    UpdateWindow (hwndFrame);               // Update window

/** MDICREATESTRUCT variables *****/
    mcs.szTitle = (LPSTR)"Child0";           // MDI child window's title
```

```
mcs.szClass = "Child";                                // MDI child window's class name
mcs.hOwner   = hInstance;                             // Instance handle
mcs.x        = mcs.cx = 0;                            // x = MDI child window's
                                                       // upper-left x position;
                                                       // cx = MDI child window's
                                                       // initial width
mcs.y        = mcs.cy = 0;                            // y = MDI child window's
                                                       // upper-left y position;
                                                       // cy = MDI child window's
                                                       // initial height
mcs.style    = WS_MAXIMIZE;                          // Additional MDI child window
                                                       // styles (maximize window)

hwndChild   = (WORD)SendMessage (hwndMDIClient,      // Generate and send
                               WM_MDICREATE,        // message to child
                               0,                   // window
                               (LONG)
                               (LPMDICREATESTRUCT)
                               &mcs);

while (GetMessage (&msg, NULL, 0, 0))      // Message reading
{
    if ( !TranslateMDISysAccel (hwndMDIClient, &msg) &&
        !TranslateAccelerator (hwndFrame, hAccel, &msg))
        // Read accelerator data

    {
        TranslateMessage (&msg); // Message translation
        DispatchMessage (&msg); // Send message to Windows
    }
}
return 0;
}

BOOL MDIInit (HANDLE hInstance)                      // Initialize instance handle
{
/** Specify window classes *****/
WNDCLASS  wcMDIClass;                                // Main window class

wcMDIClass.style         = 0;                         // Horizontal & vertical
                                                       // redraw of client area
wcMDIClass.lpfnWndProc = FrameWndProc;              // Window function
wcMDIClass.cbClsExtra   = 0;                         // No extra bytes
wcMDIClass.cbWndExtra   = 0;                         // No extra bytes
wcMDIClass.hInstance    = hInstance;                  // Instance
wcMDIClass.hIcon        = LoadIcon(NULL, IDI_APPLICATION); // Icon
wcMDIClass.hCursor       = LoadCursor(NULL, IDC_ARROW); // Cursor
```

```
wcMDIClass.hbrBackground = GetStockObject(WHITE_BRUSH); // Background
wcMDIClass.lpszMenuName = "MDIMenu"; // Menu
wcMDIClass.lpszClassName = "Frame"; // Window class

if (!RegisterClass (&wcMDIClass) ) // Register window class
    return FALSE; // Return FALSE if registration fails

/** Child window information *****/
wcMDIClass.lpfnWndProc = MDIChildWndProc; // Window function
wcMDIClass.hIcon = LoadIcon(NULL, IDI_HAND); // Icon
wcMDIClass.lpszMenuName = NULL; // No menu
wcMDIClass.cbWndExtra = 4; // Extra bytes
wcMDIClass.lpszClassName = "Child"; // Window class

if (!RegisterClass(&wcMDIClass)) // Register window class
    return FALSE; // Return FALSE if registration fails
    // If registration is successful,
return TRUE; // Return TRUE
}

/** FrameWndProc *****/
/** Main window function */
/** */

LONG FAR PASCAL FrameWndProc ( hwnd, message, wParam, lParam )
HWND      hwnd; // Window handle
unsigned   message; // Message type
WORD      wParam; // Message-dependent 16 bit value
LONG      lParam; // Message-dependent 32 bit value
{
    CLIENTCREATESTRUCT ccs; // Client creation structure
    MDICREATESTRUCT    mcs; // MDI creation structure
    HWND             hWndChild; // Handle to child window(s)
    static HANDLE     hInst; // Instance handle
    static int        i = 1; // i variable
    char            wntitle[10]; // Window title
    static HMENU      hMenu1, hMenu2; // Menu handles

    switch (message) // Process messages
    {
        case WM_CREATE: // Create window
            hInst = GetWindowWord(hwnd, GWW_HINSTANCE);
            hMenu2 = LoadMenu(hInst, "MDI2Menu"); // Second menu
            hMenu1 = GetMenu(hwnd); // First menu
            ccs.hWindowMenu = GetSubMenu (hMenu1,0);
            ccs.idFirstChild = ID_FIRST;
```

```
/** Specify appearance of applications child windows *****/
hwndMDIClient =
    CreateWindow ("mdiclient",           // Window class name
                  NULL,                 // Window caption
                  WS_CHILD|              // Child window
                  WS_CLIPCHILDREN| // Repaint main window
                                      // without deleting
                                      // child windows
                  WS_VSCROLL|           // Vertical scroll bar
                  WS_HSCROLL,           // Horizontal scroll bar
                  0, 0,                 // Upper-left x and y
                  0, 0,                 // Initial x and y sizes
                  hwnd,                 // Parent window
                  NULL,                 // No menu
                  hInst,                // Application instance
                  (LPSTR)&ccs);       // Creation parameters

ShowWindow (hwndMDIClient,
            SW_SHOW); // Make window visible
break;

case WM_COMMAND:           // Menu and command access
    switch (wParam)
    {
        case ID_NEW: // [New Window] menu item
                      // Specifies new win. parameter
            wsprintf(wntitle, "Child%i", i++);
            mcs.szTitle      = (LPSTR)wntitle;
            mcs.szClass     = "Child";
            mcs.hOwner      = hInst;
            mcs.x = mcs.cx = CW_USEDEFAULT;
            mcs.y = mcs.cy = CW_USEDEFAULT;
            mcs.style       = 0;

            hWndChild = (WORD)SendMessage
                         (hwndMDIClient,
                          WM_MDICREATE,
                          0,
                          (LONG)
                          (LPMDICREATESTRUCT)&mcs);
            break;

        case ID_TILE:   // [Tile] menu item
            SendMessage (hwndMDIClient, // Send msg
                         WM_MDITILE, // to tile
                         0,           // existing
                         0L);         // clients
    }
}
```

```
        break;

    case ID_CASC: // [Cascade] menu item
        SendMessage (hwndMDIClient, // Send msg
                     WM_MDICASCADE, // cascade
                     0,                // existing
                     0L);              // clients
        break;

    case ID_ICON: // [Arrange Icons] menu item
        SendMessage (hwndMDIClient, // Send msg
                     WM_MDIICONARRANGE, // to
                     0,                  // arrange
                     0L);              // minimized
        break;                      // icons

    case ID_MAXI: // [Maximize] menu item
        hWndChild = LOWORD(SendMessage
                            (hwndMDIClient,
                             WM_MDIGETACTIVE,
                             0,
                             0L));
        SendMessage (hwndMDIClient,
                     WM_MDIMAXIMIZE,
                     hWndChild,
                     0L);
        break;

    case ID_REST: // [Restore] menu item
        hWndChild = LOWORD(SendMessage
                            (hwndMDIClient,
                             WM_MDIGETACTIVE,
                             0,
                             0L));
        SendMessage (hwndMDIClient,
                     WM_MDIRESTORE,
                     hWndChild,
                     0L);
        break;

    case ID_NEXT: // [Next] menu item
        SendMessage (hwndMDIClient,
                     WM_MDINEXT,
                     0,
                     0L);
        break;
```

```
case ID_MENU1: // [First Menu] menu item
    SendMessage (hwndMDIClient,
                 WM_MDISETMENU,
                 0,
                 MAKELONG(hMenu2,
                           GetSubMenu
                           (hMenu2, 0)));
    DrawMenuBar(hwnd);
    break;

case ID_MENU2: // [Toggle] menu item
    SendMessage (hwndMDIClient,
                 WM_MDISETMENU,
                 0,
                 MAKELONG(hMenu1,
                           GetSubMenu
                           (hMenu1, 0)));
    DrawMenuBar(hwnd);
    break;

case ID_CLOSE: // [Close] menu item

    ShowWindow(hwndMDIClient, SW_HIDE);
    // Hide client window to avoid
    // unnecessary repainting

    while (hWndChild = GetWindow
           (hwndMDIClient,
            GW_CHILD))
        // Search for MDI child windows
        // and destroy using WM_MDIDESTROY
    {
        while(hWndChild &&
              GetWindow(hWndChild,
                         GW_OWNER))
            // Handle for an icon instead?

        {
            hWndChild = GetWindow
                (hWndChild,
                 GW_HWNDNEXT);
            // Pass it on and remove icon
        }
        if(hWndChild)
        {
            SendMessage (hwndMDIClient,
                        WM_MDIDESTROY,
```

```
        (WORD) hWndChild,
        0L);
    // Window? Destroy window
}
else break;
}
ShowWindow( hwndMDIClient, SW_SHOW);
// Make client window visible again
break;

default:
    DefFrameProc(hwnd,
        hwndMDIClient,
        WM_COMMAND,
        wParam,
        0L);
}
break;

case WM_DESTROY:           // End
    PostQuitMessage (0);
    break;

default:
    return DefFrameProc (hwnd,
        hwndMDIClient,
        message,
        wParam,
        lParam);
}

return 0;
}

/** Child window function *****/
LONG FAR PASCAL MDIChildWndProc ( hwnd, message, wParam, lParam )
HWND      hwnd;           // Window handle
unsigned   message;        // Message type
WORD      wParam;         // Message-dependent 16 bit value
LONG      lParam;          // Message-dependent 32 bit value
{
    HANDLE  hInst;          // Instance handle
    HWND    hwndList, hwndFrame; // Window list and frame handles
    WORD    Index;           // Index variable

    switch (message)        // Process messages
    {
```

```
case WM_CREATE:          // Create window
    hInst = GetWindowWord(hwnd, GWW_HINSTANCE);
    hwndList = CreateWindow ("listbox",   // Win class name
                            NULL,           // Caption
                            WS_CHILD|        // Child window
                            WS_VISIBLE|      // styles
                            LBS_STANDARD,    // Common styles
                            0,               // Upper-left x
                            0,               // Upper-left y
                            0,               // Initial x
                            0,               // Initial y
                            hwnd,            // Parent window
                            ID_LIST,         // Menu used
                            hInst,           // App. instance
                            NULL);          // No creation parameters

/** List directory and set focus to this window *****/
    DlgDirList (hwnd, "*.*", ID_LIST, 0,0x10 | 0x4000);
    SetWindowWord (hwnd, 0, (WORD)hwndList);
    SetWindowWord (hwnd, 2, (WORD)0);
    SetFocus (hwndList); // Set input focus
    break;

case WM_MDIACTIVATE:     // MDI activation message
    if (wParam)
    {
        Index = GetWindowWord (hwnd, 2);
        SetWindowWord (hwnd, 2, (WORD)Index+1);
        hwndList = GetWindowWord (hwnd, 0);
        SendMessage (hwndList,
                     LB_SETCURSEL,
                     Index+1,
                     0L);
    }
    break;

case WM_SIZE:             // Sizing message
    hwndList = GetWindowWord (hwnd, 0); // Get control ID
    MoveWindow (hwndList, 0, 0, LOWORD(lParam),
                HIWORD(lParam), TRUE); // Move controls
    return DefMDIChildProc (hwnd,      // Pass unprocessed
                           message,   // messages to this
                           wParam,    // function
                           lParam);
    break;
```

```
case WM_SETFOCUS:           // Set input focus
    hwndList= GetWindowWord (hwnd, 0);
    SetFocus (hwndList);
    break;

case WM_COMMAND:           // Commands
    switch (wParam)
    {
        case ID_LIST:// List menu
            Index = (WORD)SendMessage(GetWindowWord
                (hwnd, 0),
                LB_GETCURSEL,
                0,
                0L);
            SetWindowWord(hwnd, 2, (WORD)Index);
            break;

        default:
            break;
    }
    break;

default:
    return DefMDIChildProc (hwnd, message, wParam, lParam);
}
return FALSE;
}
```

## Module definition file: MDI.DEF

```
NAME      MDI
DESCRIPTION 'MDI example'

EXETYPE   WINDOWS
STUB      'WINSTUB.EXE'

CODE      PRELOAD MOVEABLE DISCARDABLE
DATA      PRELOAD MOVEABLE MULTIPLE

HEAPSIZE  2048
STACKSIZE 8192

EXPORTS   FrameWndProc
          MDIChildWndProc
```

## Resource script: MDI.RC

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

MDIMenu MENU
BEGIN
    POPUP "&Window"
    BEGIN
        MENUITEM "&New Window\tShift+F8", ID_NEW
        MENUITEM "New &Menu", ID_MENU1
    END

    POPUP "&All"
    BEGIN
        MENUITEM "&Tile\tCtrl+T", ID_TILE
        MENUITEM "&Cascade\tCtrl+C", ID_CASC
        MENUITEM "&Arrange Icons", ID_ICON
        MENUITEM "C&lose", ID_CLOSE
    END

    POPUP "&Single"
    BEGIN
        MENUITEM "&Maximize",
        MENUITEM "&Restore",
        MENUITEM "&Next",
    END
END

MDI2Menu MENU
BEGIN
    POPUP "&Toggle"
    BEGIN
        MENUITEM "New &Menu", ID_MENU2
    END
END

MDIMenu ACCELERATORS
BEGIN
    VK_F8, ID_NEW, SHIFT, VIRTKEY
    "^T", ID_TILE
    "^C", ID_CASC
END
```

## **Header file: MDI.H**

```
#define ID_FIRST 10  
  
#define ID_LIST 100  
  
#define ID_TILE 100  
#define ID_CASC 101  
#define ID_ICON 102  
#define ID_CLOSE 103  
  
#define ID_NEW 200  
#define ID_MENU1 201  
#define ID_MENU2 202  
  
#define ID_MAXI 300  
#define ID_REST 301  
#define ID_NEXT 302
```

To compile and link this application faster, create a batch file named RCOMPILE.BAT. You can create this file using the DOS COPY CON command, any text editor or word processor that generates ASCII files. Type the following to create RCOMPILE.BAT:

```
cl -c -Gw -Zp %1  
rc -r %1.rc  
link /align:16 %1,%1.exe,,libw+slibcew,%1.def  
rc %1.res
```

Save this file to a directory contained in your path or the directory containing your source, module definition and resource scripts. Type the following and press e to compile MDI:

```
rcompile mdi
```

The batch file performs all the tasks needed.

## **How MDI.EXE works**

Two classes are registered in the MDIInit initialization routine: one for the frame window and one for all child windows. The frame window has a menu called "MDIMenu". The cbWndExtra field is filled with the value 4 in order to reserve four additional bytes per window. You can

use any names for the two class names. In our example, we used "Frame" for the frame window and "Child" for the child windows.

The CreateWindow function, which creates the frame window with the style **WS\_CLIPCHILDREN**, generates a **WM\_CREATE** message that is sent directly to its window function. The handles of the instance, the set menu, and the second menu are obtained on the basis of this message.

Then the **ccs** variable of the **CLIENTCREATESTRUCT** structure can be filled with the handle of the pop-up menu, under which the title of the child window is supposed to appear, and with the **ID** value of the first child window.

The CreateWindow function needs the **css** variable when it creates an MDI client window. Since the frame window is the parent window, you specify the **WS\_CHILD** style parameter and the window handle of the frame window.

In addition, the **WS\_CLIPCHILDREN** and **WS\_VSCROLL** styles are set for the scroll bars. Since the client window doesn't have its own client area, you can use the **ShowWindow** function to make this window visible. After the **WM\_CREATE** message is processed, you jump back to the **WinMain** function.

If one of the two windows couldn't be created, the entire MDI application ends. Otherwise, the frame window is displayed and its accelerator table is loaded. Since the first child window is supposed to appear in the frame window immediately, you must create it before entering the message loop.

To do this, fill the **mcs** variable of the **MDICREATESTRUCT** structure with values. The initial position and size are both set to 0 because the window is being displayed by the **WS\_MAXIMIZE** style in its maximum size. The **WM\_MDICREATE** message of the supplied variable (**mcs**) is sent to the client window so that the client window creates a MDI child window.

The client window sends the **WM\_CREATE** message to the window function of the child window. Since the MDI child window is entirely overlapped by a list box, we use the **CreateWindow** function to create another window that is one level lower than the MDI child window and

has the predefined class of "list box". The DlgDirList function fills this list box with all of the files of the current directory.

To be able to access the handle of the list box and its index, which specifies the selected string, the SetWindowWord function writes both of these values to the four extra bytes. Also, the focus is passed to the list box window. The MoveWindow function doesn't set the list box to the size of its parent window until the WM\_SIZE message. The GetWindowWord function retrieves the necessary handle of the list box from the extra bytes. Then the WM\_SIZE message still has to be passed to the default routine DefMDIChildProc.

After the first MDI child window was created, WinMain branches to the message loop. Each message is checked for an accelerator key for the frame window or the child window. If the message contains an accelerator key, it jumps to the appropriate window function immediately.

The frame window has a menu that contains the pop-up menus Window, All, and Single, which have different menu items. The active MDI child window can also be resized via the Maximize and Restore menu items. Either the WM\_MDIMAXIMIZE or WM\_MDIRESTORE message is sent to the client window for this purpose.

To get to the next MDI child window, you can use either the accelerator (**[Ctrl]** + **F6**) or the Next menu item. Both send the WM\_MDINEXT message to the parent window. Two of the menu items in the All pop-up menu affect all existing MDI child windows. The windows can either overlap with the help of the WM\_MDICASCADE message or be arranged next to each other with the WM\_MDITILE message. The third item rearranges all of the icons by sending the WM\_MDIICONARRANGE message to the client window.

## Closing all windows and icons

If you want to close all child windows by using the Close menu item from the All menu, the client window disappears during the process with the help of the ShowWindow function and the SW\_HIDE value. This helps prevent the existing child windows from appearing each time the child window above is destroyed.

The client window reappears when there are no more child windows. The deletion takes place in a while loop. As long as the GetWindow function returns a valid handle of a child window, this handle is passed to the client window in the wParam parameter during the WM\_MDIDESTROY message.

In addition to normal child windows, the **Close** menu item also checks for child windows appearing as minimized icons. The routine then closes these windows as well.

You can use the **New Window** menu item to create as many MDI child windows as you want. To do this, a variable of the MDICREATESTRUCT structure must be supplied with data, just as you did at the beginning. Along with the name "Child", the title consists of an ascending number that is used to mark the individual child windows.

Along with the menu items we just mentioned, the **Window** pop-up menu also has another menu item called **New Menu**. When the user selects this item, the entire "MDIMenu" is replaced by "MDI2Menu", which is also defined in the .RC file.

This new menu consists of only one pop-up menu named **Toggle**, with a menu item (**New Menu**) that you can use to retrieve the original status. You use the WM\_MDISETMENU message to exchange the menus of the frame window. This message must be sent to the client window. The new menu handle is passed in the low-order word and the handle of the pop-up menu is passed in the high-order word.

The last menu handle we mentioned is needed so that Windows knows to which pop-up menu it should add the title of the MDI child. If the handle is omitted, a title doesn't appear. Each time you exchange menus, you must redraw the menu bar with the DrawMenuBar function.

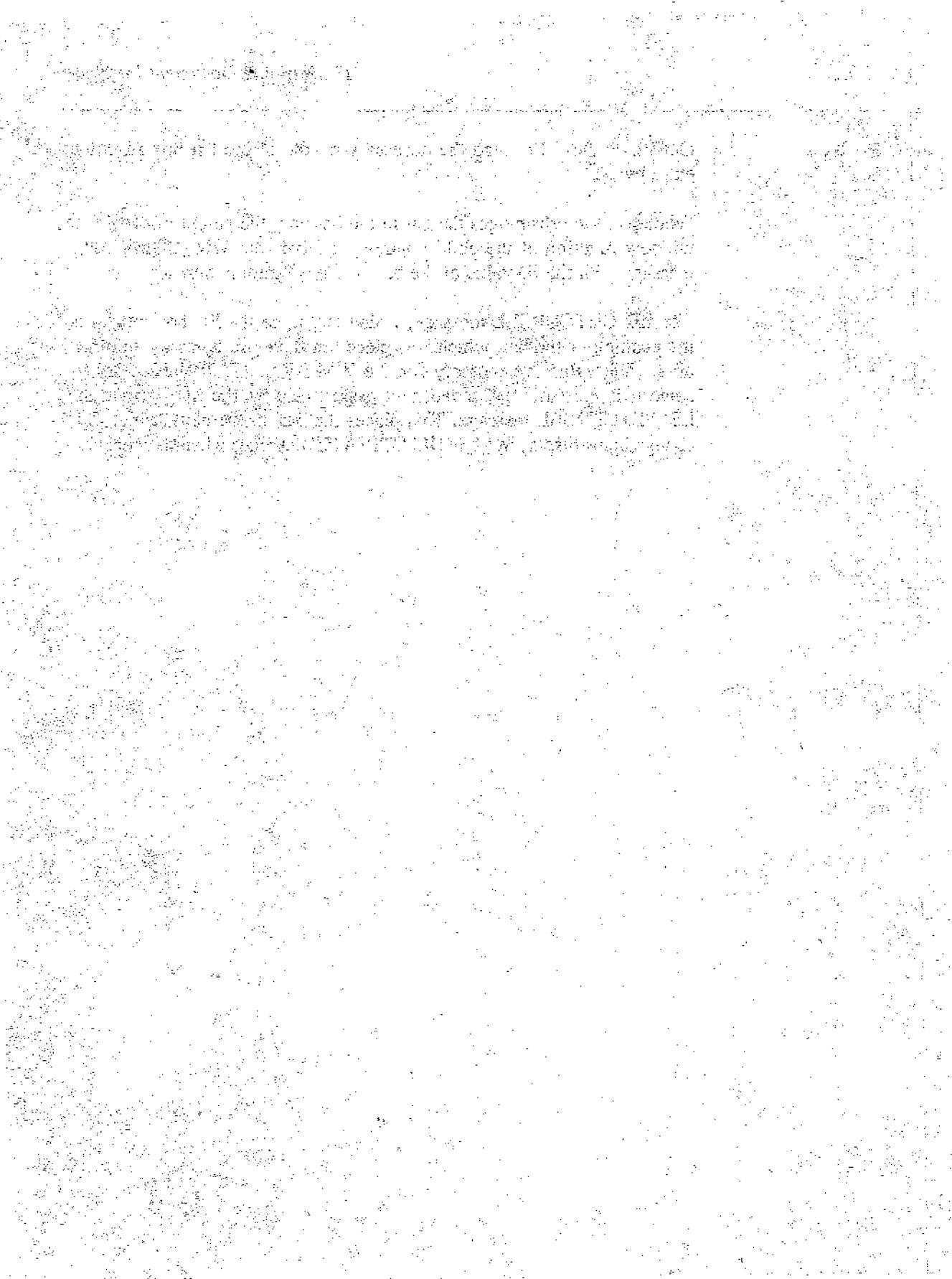
Besides the WM\_CREATE and WM\_SIZE messages, three other messages are processed in the window function of the MDI child window. As soon as a child window is enabled, it receives the focus. In the sample application, the list box of the active window is supposed to have the focus so that you can use the cursor keys to immediately select another string without having to click the box with the mouse.

Therefore, the focus is passed to the list box during the WM\_SETFOCUS message with the help of the SetFocus function. Use

GetWindowWord to read the necessary handle of the list box from the extra bytes.

Whenever the user moves the bar in the list box, the parent window (the window function of the child window) receives the WM\_COMMAND message with the ID value of the box in the wParam parameter.

The LB\_GETCURSEL message, which is sent to the list box, retrieves the position of the bar, which was placed in the extra bytes by an offset of 2. This value is necessary for the WM\_MDIActivate message, since it is incremented, saved, and passed back to the list box via the LB\_SETCURSEL message. This places the bar on the next string. This change helps display WM\_MDIActivate messages when they occur.



# Index

-Alnw compiler switch ..... 350  
 -Asnw compiler switch ..... 350  
 \$ footnote character ..... 379  
 &ps parameter ..... 78  
 &ps.rcPaint ..... 88  
 &rect ..... 88  
 + footnote character ..... 381  
 -c compiler switch ..... 18  
 -Gs compiler switch ..... 18  
 -Gw compiler switch ..... 18, 340  
 -r compiler switch ..... 130  
 -Zp compiler switch ..... 18  
 [ALIAS] section, Help project file ..... 389  
 [BUILDTAGS] section, Help project file ..... 388  
 [FILES] section, Help project file ..... 386  
 [MAP] section, Help project file ..... 388  
 [OPTIONS] section, Help project file ..... 386  
 \a control character ..... 155  
 \_lclose ..... 309  
 \_lcreat ..... 304  
 \_llseek ..... 307, 309, 321  
 \_lopen ..... 304  
 \_lread ..... 307  
 /align:16 compiler switch ..... 18

## A

Accelerator keys ..... 173  
 ACCELERATORS table ..... 173  
 ALIGN all dialog boxes ..... 26  
 ALTERNATE fill mode ..... 97, 114  
 ALTERNATE, SetPolyFillMode ..... 98  
 Annotate...item, Help application ..... 374  
 ANSI character set ..... 46  
 AnsiToOem ..... 304  
 AppendMenu ..... 158, 159, 161, 171  
 Application message queue ..... 45  
 Arc function ..... 90, 97

## B

Background mode ..... 73  
 Basic graphic functions ..... 89  
 Basic text output functions ..... 79  
 BeginPaint ..... 74, 75, 78  
 bErase ..... 77  
 biBitCount variable ..... 234  
 Binary Raster Operation ..... 114  
 BitBlt ..... 220, 225, 232, 239, 279  
 BITMAP ..... 151, 155  
 Bitmap format ..... 278, 280  
 BITMAP resource script statement ..... 216  
 Bitmap, creating as a resource ..... 216  
 Bitmap, obtaining data ..... 279  
 Bitmap, obtaining from clipboard ..... 279  
 Bitmap, writing to clipboard ..... 278  
 BITMAPCOREINFO structure ..... 235  
 BITMAPFILEHEADER ..... 233  
 BITMAPINFO ..... 239  
 BITMAPINFOHEADER structure ..... 233  
 Bitmaps ..... 151, 215  
 Bitmaps, creating ..... 151  
 Bitmaps, creating with a bit array ..... 218  
 Bitmaps, creating with GDI functions ..... 218  
 BLACKNESS ..... 223  
 BLACKONWHITE mode ..... 222  
 bLERI global variable ..... 196  
 bm variable ..... 231  
 BM\_GETCHECK message ..... 186, 196  
 BM\_SETCHECK message ..... 186, 195  
 bmiColors array ..... 235  
 BMP extension ..... 151  
 bMyBitmap Boolean variable ..... 291  
 bMyData variable ..... 277  
 BN\_DOUBLECLICKED code ..... 176  
 Bookmark menu, Help application ..... 375  
 Bookmarks, in Help ..... 375

---

Boolean operation .....	114	Chord function .....	97
bRILE global variable .....	196	chText variable .....	276
Browse sequence, Help system .....	380	chtmName string array .....	127
Brush .....	106	Client area .....	11
Brushes, creating your own .....	107	Client area, changing size .....	116
Brushes, types .....	107	Client area, determining size .....	116
BS_AUTOCHECKBOX styles .....	175	Client window, MDI .....	406
BS_AUTORADIOBUTTON style .....	175	CLIENTCREATESTRUCT ....	407, 414, 426
BS_CHECKBOX .....	175	ClientToScreen .....	56, 161, 172
BS_GROUPBOX .....	175	Clipboard .....	265
BS_PUSHBUTTON .....	175	Clipboard owner .....	293
BS_RADIOBUTTON .....	175	Clipboard viewer .....	294, 295
bSystemExit parameter .....	348	Clipboard viewer, example of .....	297
bText Boolean variable .....	291	Clipboard, formats .....	266
Build option, Help project file .....	387	CLIPBRD.EXE application .....	265, 295
BUILDTAGS .....	384	Clipping region .....	76
BUTTON .....	175	CloseClipboard .....	267, 269, 280
Button NC .....	176	Closing a file .....	306
<b>C</b>			
C compiler, compiling a DLL .....	350	CLPBRD.EXE .....	294
Callback functions .....	342	CmdShow variable .....	27
Capture .....	28	CODE .....	36
CB_DIR message .....	186, 195	Color bitmaps .....	216
CB_ERR .....	196	Color index .....	89
CB_SELECTSTRING .....	186	Color planes .....	216
cbWndExtra field .....	408, 425	COLORONCOLOR mode .....	222
ccs variable .....	426	COLORREF .....	106
Centering text .....	83	Combo box .....	185
CF_BITMAP identifier .....	278	COMBOBOX .....	175
CF_OWNERDISPLAY parameter .....	294	Combobox NC .....	176
CF_TEXT .....	266	Common device context .....	74
Chained task list .....	30	Compiler switches .....	18, 340, 350
ChangeClipboardChain .....	296, 298	Compiling .....	18
Character cell .....	117	Context string, Help system .....	378
CHECKED option .....	155	Context-sensitive Help .....	388
Checkmarks, defining .....	159	Control .....	174
CheckMenuItem .....	158, 161, 172	Control Panel .....	323
CheckRadioButton .....	177	Coordinate origin .....	75
Child window, creating .....	25, 26	Copy menu item .....	277
Child window, MDI .....	407	Copy menu, Help application .....	374
CountClipboardFormats .....	292	CreateBitmap .....	218, 225

---

CreateBrushIndirect .....	107	Delayed rendering technique .....	293
CreateCompatibleBitmap .....	218, 291	DeleteDC .....	225, 231
CreateCompatibleDC .....	225, 231, 279	DeleteMenu .....	158, 161
CreateDialog .....	184	DeleteObject .....	104, 224
CreateDIBitmap .....	236, 239	DestroyMenu .....	158, 161, 172
CreateDiscardableBitmap .....	218	DestroyWindow .....	34, 183
CreateFont .....	117	Device context .....	71
CreateFontIndirect .....	117	Device context attributes .....	72
CreateMenu .....	158	Device contexts, types .....	73
CreatePen .....	104	Device-dependent bitmap .....	215
CreatePenIndirect .....	104	Device-dependent bitmap, creating .....	215
CreatePopupMenu .....	159, 161	Device-dependent bitmap, example of .....	225
CreatePushB .....	370	Device-dependent mode .....	200
CreateWindow .....	29, 156, 409, 426	Device-independent bitmap (DIB) ....	215, 232
Creating a Help system .....	375	Device-independent bitmap, example of .....	238
Cross reference, Help system .....	382	Device-independent bitmap, functions .....	235
CS_CLASSDC style .....	74	Device-independent bitmap, structure .....	232
CS_DBLCLKS flag .....	56	Dialog box .....	174
CS_HREDRAW .....	76, 77, 87	Dialog box, calling .....	183
CS_VREDRAW .....	76, 77, 87	Dialog box, creating .....	178
CUR extension .....	144	Dialog box, removing from screen .....	183
Cursor .....	143	Dialog box, types .....	181
Cursor, creating .....	144	Dialog routine .....	181
Custom controls .....	174	DIALOG.EXE utility application .....	178
Custom modes .....	203	DialogBox .....	184, 186
Cut menu item .....	277	DialogbxWndProc window procedure ....	194
CW_USEDEFAULT value .....	410	DIB .....	215
cx, MDICREATESTRUCT .....	410	Digital Differential Analyzer .....	90
cy, MDICREATESTRUCT .....	410	DISCARDABLE parameter .....	36
<b>D</b>		DispatchMessage .....	30, 31, 33, 45
DATA statement .....	36, 349	Display context .....	73
DataBase .....	40	Display context, types .....	74
DDA .....	90	DISPLAY.DRV .....	339
Default device context .....	74	DLG file, example .....	179
Default values, device context .....	72	DlgDirList .....	195, 414, 427
Default window functions .....	32	DLL function access, example .....	362
DefFrameProc routine .....	405	DLL resource, access example .....	353
DefMDIChildProc .....	412, 413	DLL, adding to an application .....	351
DefWindowProc .....	32	DLL, compiling with C compiler .....	350
DefWndProc .....	44	DLL, creating .....	347
		DLL, examples .....	353

DLL, resources ..... 348  
DLLCEW.LIB ..... 350  
DLLFCT1 function ..... 351  
DLLs ..... 10, 38  
Double-click ..... 55  
DPtoLP ..... 202  
Drawing mode ..... 114  
DrawMenuBar ..... 414, 428  
DrawText ..... 79, 83, 88  
dsaddress value ..... 343  
DT\_CENTER ..... 83  
DT\_SINGLINE flag ..... 88  
DT\_VCENTER ..... 83, 88  
DT\_WORDBREAK ..... 83  
dwData parameter ..... 391  
dwUsage parameter ..... 236  
Dynamic Link Libraries ..... 10, 38

## **E**

EDIT ..... 175  
Ellipses ..... 96  
EM\_LIMITTEXT message ..... 308, 319  
EmptyClipboard ..... 267, 269, 292  
EnableMenuItem ..... 158, 161  
EndDialog ..... 183, 186, 196  
EndPaint ..... 74, 75, 78, 79  
Endpoint ..... 95  
Enhanced mode ..... 7  
Entry table ..... 40  
EnumClipboardFormats ..... 292  
Epilog code ..... 340  
EXPORTS statement ..... 38, 197

## **F**

FAR function ..... 340  
FAR PASCAL ..... 32  
FARPROC data type ..... 183  
FARPROC pointer ..... 194  
fErase flag ..... 78  
fgbReserved[16] ..... 78

Field ..... 78  
File menu, Help application ..... 374  
File pointer, setting ..... 307  
File, closing ..... 306  
File, opening ..... 304  
File, reading ..... 306  
File, writing to ..... 307  
FillRect ..... 362, 371  
fIncUpdate ..... 78  
fixed pitch ..... 117  
FNT extension ..... 151  
Focus ..... 29  
Font ..... 115, 151  
Font resource ..... 119  
Font table, in GDI ..... 119  
FONTEDIT.EXE utility ..... 151  
Fonts ..... 151  
Fonts, creating ..... 151  
Formats, clipboard ..... 266  
Formats, storing two in clipboard ..... 292  
Formatting text ..... 83  
Frame window, MDI application ..... 405  
Free pop-up menus, creating ..... 159  
FreeLibrary ..... 352, 354  
FrProcInstance ..... 183, 186  
fRestore ..... 78  
Functions ..... 10

## **G**

GDI ..... 71, 89  
GDI.EXE ..... 10, 38  
GetBitmapBits ..... 223, 238, 239  
GetClientRect ..... 87, 95, 116  
GetClipboardData ..... 266, 268, 269, 277, ..... 279, 293  
GetCurrentPosition ..... 90, 95  
GetDC ..... 74, 75  
GetDeviceCaps ..... 218  
GetDIBits ..... 237, 239  
GetDlgItem ..... 176, 186, 195  
GetDlgItemText ..... 186, 196

---

GetFocus .....	44	HBRUSH .....	20, 104
GetMenu .....	161	hClipData .....	267, 277
GetMenuCheckMarkDimensions ....	160, 161	HCURSOR .....	20
GetMenuItem .....	161, 171	HDC .....	20, 78, 79
GetMessage .....	30, 33, 45	HEAPSIZE statement .....	38, 349
GetObject .....	225, 231, 233, 261, 279	Help compiler, calling .....	390
GetPixel .....	223	Help project file .....	385
GetPrivateProfileInt .....	326	Help system .....	373
GetPrivateProfileString .....	326	Help system, creating .....	373
GetProfileInt .....	324	Help system, structure .....	376
GetProfileString .....	323, 324, 326, 336	Help text .....	374
GetScrollPos .....	254, 262	Help topic files .....	378
GetStockObject .....	23, 103, 107, 116	Help window .....	373
GetSubMenu .....	161, 407	HELP_CONTEXT .....	391
GetSysColor .....	89, 362	HELP_INDEX .....	391
GetSystemMetrics .....	54, 248	HELP_KEY .....	391
GetTempFilename .....	303	hFile parameter .....	306, 307
GetTextMetrics .....	115, 127, 199	HFONT .....	20, 104
GetWindow .....	414	HICON .....	20
GetWindowLong .....	408	hIcon field .....	408
GetWindowText .....	186	HiliteMenuItem .....	158
GetWindowWord .....	150, 261, 320, 408, ..... 427, 429	hInstance .....	21, 32
Global memory area .....	267	HIWORD macro .....	55
Global memory blocks, releasing .....	277	HMENU .....	20
GlobalAlloc .....	269, 277	hMyData handle .....	267, 277
GlobalFree .....	269, 278	hOwner field .....	410
GlobalLock .....	267, 269	hOwner, MDICREATESTRUCT .....	410
GlobalReAlloc .....	269, 278	HPALETTE .....	20
GlobalSize .....	269, 277	HPEN .....	20, 104
GlobalUnlock .....	269, 277	hPrevInstance .....	21
Graphic function .....	79	HRGN .....	20
Graphics Device Interface .....	10	hSrcDC handle .....	220
GRAYED option .....	155	HSTR .....	20
<b>H</b>			
Handle .....	265	Hungarian notation .....	9
Hardware interrupt .....	43	hWindowMenu .....	407
HBITMAP .....	20	hwnd .....	29, 77
HBITMAP CreateBitmap .....	216	hWnd handle .....	31
HBITMAP CreateBitmapIndirect .....	216	hWnd parameter .....	32, 78

**I**

ICO file ..... 132  
Icon ..... 131  
ICON keyword ..... 133  
Icons, creating ..... 132  
IDC\_ARROW ..... 23  
idFirstChild ..... 407  
IDI\_APPLICATION ..... 23, 132  
IDI\_ASTERISK ..... 132  
IDI\_EXCLAMATION ..... 132  
IDI\_HAND ..... 132  
IDI\_QUESTION ..... 132  
Import library ..... 39, 350  
Imported names table ..... 40  
IMPORTS statement ..... 38, 339, 351  
INACTIVE option ..... 155  
Index option, Help project file ..... 387  
Initialization ..... 22  
Initialization file, example ..... 326  
Initialization files ..... 321, 322  
Initialization files, creating ..... 325  
Input messages ..... 43  
InsertMenu ..... 158, 161  
Instance handles ..... 21  
InstanceThunk ..... 343  
Integer value ..... 38  
InvalidateRect ..... 54, 77, 212, 252, 371  
InvalidateRgn ..... 77  
IsClipboardFormatAvailable ..... 268, 269, 277, ..... 291  
IsDialogMessage ..... 184  
Isotropic ..... 203

**K - L**

KERNEL.EXE ..... 10, 38  
Keyboard support, scroll bars ..... 251  
Keywords, Help system ..... 380  
KillTimer ..... 64, 65  
LB\_GETCURSEL message ..... 414, 429  
LB\_SETCURSEL message ..... 414, 429

Libraries ..... 10  
LIBRARY ..... 36  
LineDDA ..... 90  
LineTo ..... 90, 95  
Linker ..... 130, 350  
LISTBOX ..... 175  
Listbox NC ..... 176  
LoadAccelerators ..... 174  
LoadBitmap ..... 151, 160, 161, 216  
LoadCursor ..... 23, 144, 145  
LoadIcon ..... 23, 132, 138  
LoadLibrary ..... 352, 354  
LoadMenu ..... 161  
LoadString ..... 153, 161  
LOGBRUSH structure ..... 107, 108  
LOGFONT structure ..... 117  
Logical coordinates ..... 199  
Logical font, selecting in the DC ..... 118  
Logical fonts, creating ..... 117  
Logical operation ..... 114  
LOGPEN structure ..... 105  
LOWORD macro ..... 55  
lParam parameter ..... 30, 55, 116, 262  
lpBuffer parameter ..... 306, 307  
lpDefault variable ..... 324  
lpKeyName parameter ..... 324  
LPLOGPEN ..... 347  
lpprocDia ..... 343  
lpRect ..... 77, 347  
lpReturnedString variable ..... 324  
LPSTR ..... 19  
lpszCmdLine ..... 22  
lread ..... 309  
lstrcpy ..... 267  
lstrcpy ..... 269, 277, 326, 346  
lwrite ..... 309

**M**

Main window ..... 25, 405  
MAKEINTRESOURCE macro ..... 133  
MAKELONG ..... 362

---

MAKEPOINT macro .....	95, 172	MM_LOMETRIC .....	200
MakeProcInstance .....	183, 186, 194, 340, .....	MM_TEXT .....	200
Mapping Mode .....	199	MM_TWIPS .....	200
Mapping modes, types .....	200	Modal dialog box .....	181, 183
MAPPING.H header file .....	212	Modeless dialog box .....	181, 183
max .....	254, 262	ModifyMenu .....	158, 161, 171
MB_YESNOCANCEL .....	197	Module definition file .....	35, 349
mcs variable .....	426	Monochrome bitmap .....	224
MDI .....	403	Mouse messages .....	55
MDI application, structure of .....	403	Mouse pointer, displaying .....	54
MDI child window, styles .....	410, 411	Mouse support .....	339
MDI messages .....	408	Mouse, using with Windows applications .	54
MDICLIENT .....	175, 406	MOUSE.DRV .....	339
MDICREATESTRUCT .....	409, 414, 426	MoveTo .....	90, 95
Memory device context .....	73	MoveWindow .....	414
Memory model .....	350	MSG structure .....	27, 29
Menu .....	153	MULTIPLE .....	37
Menu name .....	155	Multiple Document Interface .....	403
Menu, changing .....	157	Multiple-line statements .....	131
Menu, linking to source text .....	156		
MenuChange item .....	170		
MENUITEM SEPARATOR line .....	155		
MENUITEM statement .....	155		
Menus, adding options .....	155		
Menus, defining in the resource script .....	153		
Message .....	30		
Message box .....	69, 197		
Message loop .....	30, 404		
Message queues .....	27		
MessageBeep .....	62		
MessageBox .....	65, 69, 197		
MF_ENABLED routine .....	278		
MF_OWNERDRAW style .....	160		
min .....	254, 262		
MM .....	205		
MM_ANISOTROPIC mode .....	203		
MM_HIENGLISH .....	200		
MM_HIMETRIC .....	200		
MM_ISOTROPIC mode .....	203		
MM_LOENGLISH .....	200		

## N

NAME .....	36
nBitCount parameter .....	218
nBits parameter .....	217
nCmdShow .....	22
NEAR function .....	340
Non-preemptive multitasking .....	30
Nonqueued message .....	33
Notification codes .....	176
nPlanes parameter .....	217, 218
NULL_PEN .....	104
Number value .....	341

## O

Octal constants .....	152
OF_REOPEN .....	321
OFSTRUCT .....	309
ofstruct variable .....	321
OpenClipboard .....	267, 269, 279
OpenFile .....	304, 309

Opening a file .....	304
Options, linking .....	155
OR operation .....	155
Ordinal number .....	38
Output functions .....	79
Output functions, device-dependent bitmaps .....	218
Output, in Windows .....	71

## **P**

PAINTBRUSH .....	232
Param parameter .....	44
PASCAL .....	20
PatBlt .....	219, 225
PATCOPY .....	223
PATCOPY raster operation code .....	220
PBRUSH.EXE .....	232
Pen .....	104
Penalty points .....	119
Physical units .....	89
Pie function .....	97
POINT array .....	90
Polygon function .....	97
Polygons, fill mode .....	98
Polyline function .....	90, 97
PolyPolygon function .....	97
Pop-up menu .....	155
Pop-up windows .....	26
POPUP statement .....	155
PostMessage .....	33
PostQuitMessage .....	34
Predefined bitmaps .....	151
PRELOAD and MOVEABLE .....	36
Private data formats .....	294
Prolog code .....	340
ps variable .....	88, 253
PS_INSIDEFRAME style .....	105
pt variable .....	30, 95

## **Q - R**

Queued message .....	33
R2_COPYPEN .....	114
Raster Operation .....	114
RASTERCAPS parameter .....	218
rcPaint .....	78
RCPP.EXE .....	130
Reading a file .....	306
Real mode .....	7
RegisterClass .....	23
RegisterClipboardFormat .....	294
ReleaseCapture function .....	56, 62
ReleaseDC .....	74, 75
Releasing global memory blocks .....	277
RemoveMenu .....	158
RESIDENTNAME statement .....	349
Resource compiler .....	129, 350
Resource DLL .....	348
Resource script .....	129
Resource segments .....	129
Resource table .....	40
Resources, DLL .....	348
Resources, types of .....	129
RestoreDC function .....	75
Return (FALSE) .....	195
RGB macro .....	106
RGB value .....	106
Root option, Help project file .....	387
ROP .....	114
ROP2 Code .....	114
RoundRect function .....	96
RTF .....	377

## **S**

SaveDC function .....	75
SB_BOTTOM message .....	250, 262
SB_ENDSCROLL parameter .....	250
SB_THUMBPOSITION message .....	250
SB_THUMBTRACK message .....	250
SB_TOP message .....	250, 262

---

Screen DLL function .....	371	SetStretchBltMode .....	222
Screen driver .....	339	SetTextAlign .....	80
ScreenToClient function .....	56	SetTextColor .....	88
Scroll bar .....	247, 406	SetTimer .....	63, 65, 230
Scroll bar, defining .....	247	Setting the file pointer .....	307
Scroll bar, example .....	254	SetViewportExt .....	205
Scroll bar, keyboard support .....	251	SetViewportOrg .....	205
Scroll bar, messages .....	249	SetWindowExt .....	205
Scroll range .....	248	SetWindowLong function .....	408
SCROLLBAR .....	175	SetWindowOrg .....	205
Scrolling .....	247	SetWindowText .....	186
ScrollWindow .....	252, 254	SetWindowWord .....	408, 414, 427
SDKPaint .....	216	ShowCursor .....	54
SDKPAINT application .....	151	ShowWindow .....	22, 27, 427
SDKPAINT.EXE ....	132, 144, 159, 230, 232	SINGLE .....	37
SelectObject .....	104, 116, 127, 279	Single-line statements .....	130
SendDlgItemMessage .....	176, 186, 195	SM_CXVSCROLL parameter .....	248
SendMessage .....	33, 251, 295	SM_CYHSCROLL parameter .....	248
Sequence numbers .....	381	SM_MOUSEPRESENT parameter .....	54
SetBitmapBits .....	218	Source code, DLL .....	347
SetBkColor .....	88	Source text, compiling .....	18
SetBkMode .....	88	SRCCOPY .....	223
SetCapture .....	29, 56, 62, 145	SRCINVERT .....	223
SetClassWord .....	73, 134, 186	Standard mode .....	7
SetClipboardData ....	269, 277, 278, 292, 293	STATIC .....	175
SetClipboardViewer .....	295, 298	StretchBlt .....	160, 221, 225, 232
SetCursor .....	145	Strikethrough character format .....	382
SetCursorPos .....	56	STRINGTABLE .....	152
SetDIBits .....	236, 237	STUB .....	36
SetDIBitsToDevice .....	237, 239	Style parameters, controls .....	175
SetDoubleClickTime .....	57	Style, MDICREATESTRUCT .....	410
SetFocus .....	44, 186, 195	sVertPos variable .....	261, 262
SetFontMapperWeight .....	120	SW_HIDE value .....	427
SetMapMode .....	200, 205	SW_SHOWMINNOACTIVE .....	27
SetMenu .....	158, 161, 170	System colors .....	88
SetMenuItemBitmaps .....	160, 161	System message queue .....	28
SetPixel .....	223	System modal dialog box .....	181
SetPolyFillMode .....	98	System Services interface .....	10
SetROP2 .....	115	SYSTEM.DRV (timer) .....	339
SetScrollPos .....	254	SYSTEM.DRV (driver) .....	63
SetScrollRange .....	248, 254	SYSTEM.INI file .....	322

szClass, MDICREATESTRUCT ..... 410  
szDataRecord string ..... 320  
szInitFileName return string ..... 336  
szTitle, MDICREATESTRUCT ..... 410

## **T**

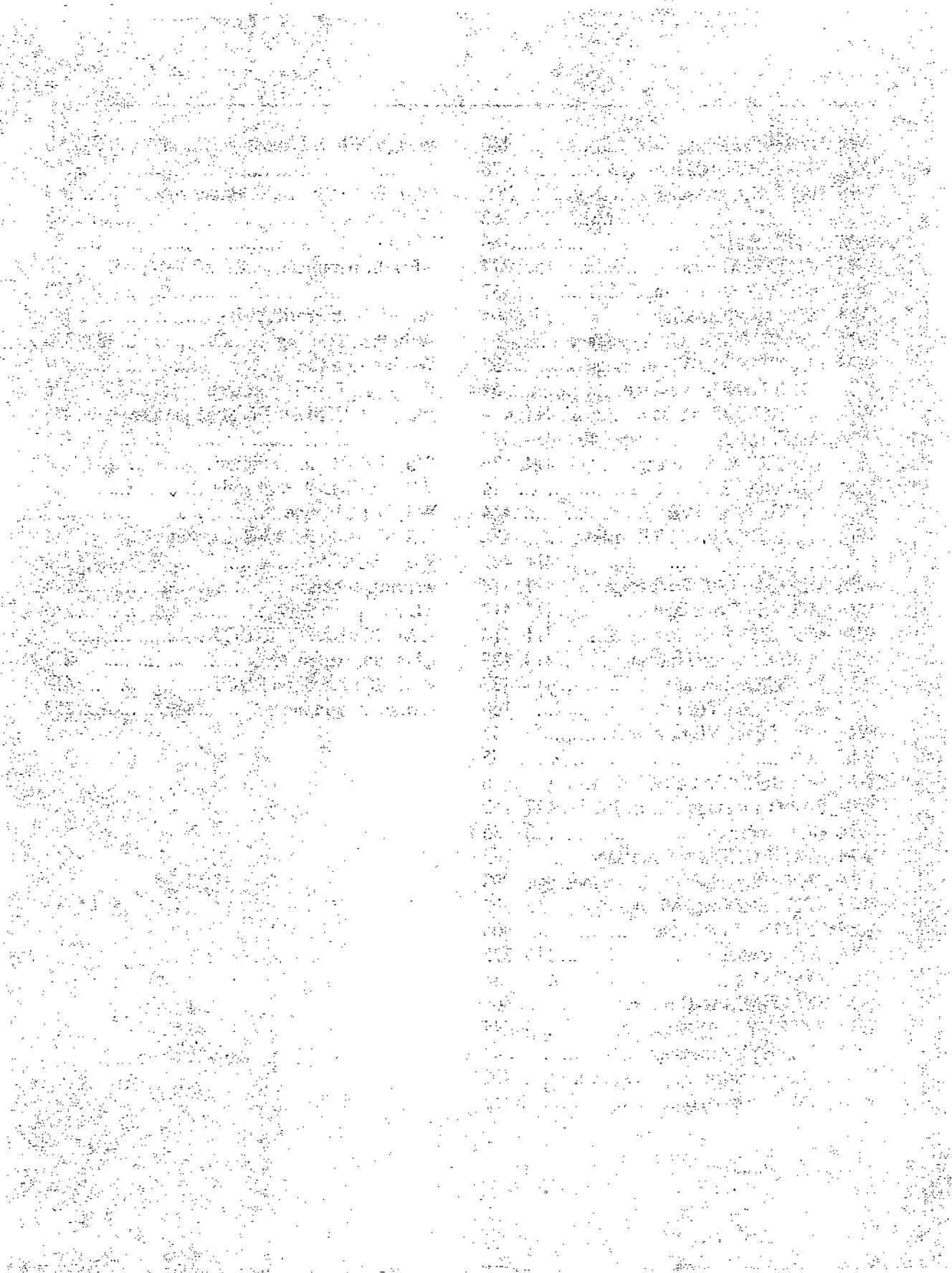
Text format ..... 266  
Text format, example of ..... 269  
Text functions ..... 79  
Text, centering ..... 83  
Text, copying to the clipboard ..... 266  
Text, formatting ..... 83  
Text, obtaining from clipboard ..... 267  
TEXTMETRIC structure ..... 115, 127  
TextOut ..... 79, 95, 127  
Thumb, of scroll bar ..... 247, 248  
Time ..... 30  
Timer messages ..... 63  
Title, Help system ..... 379  
tm variable ..... 127  
tmExternalLeading parameter ..... 116  
Topics, excluding from the Help file ..... 385  
TrackPopupMenu ..... 159, 161  
Transfer parameters ..... 31  
TranslateAccelerator ..... 174, 405  
TranslateMDISysAccel ..... 404, 414  
TranslateMessage ..... 31, 45  
typedef definitions ..... 19

## **U - Y**

Unformatted text ..... 266  
UpdateWindow function ..... 27, 76  
USER.EXE ..... 10, 28, 38  
Viewport Extents parameter ..... 199  
Viewport Origin parameter ..... 199  
Virtual Memory Manager ..... 7  
VMM ..... 7  
VpExtX ..... 200  
VpExtY ..... 200  
VpOrgX ..... 200

VpOrgY ..... 200  
WARNING option, Help project file ..... 386  
wBytes parameter ..... 307  
wcCursorClass.hCursor field ..... 150  
wCommand parameter ..... 391, 397  
WEP function ..... 348  
WHITEONBLACK mode ..... 222  
widIcon global variable ..... 196  
WIN.INI file ..... 322  
WINDING, SetPolyFillMode function ..... 98  
Window coordinates ..... 26  
Window Extents window parameter ..... 199  
Window Manager interface ..... 10  
Window Origin window parameter ..... 199  
Windows 3.0 ..... 7  
Windows 3.0, functions ..... 10  
Windows 3.0, modes ..... 7  
Windows libraries ..... 10  
Windows timer ..... 63  
WINDOWS.H header file ..... 19, 320  
WINDOWS.H include file ..... 223, 251  
WinExtX ..... 200  
WinExtY ..... 200  
WinHelp function ..... 391, 392  
WINHELP.EXE ..... 374  
WinMain ..... 13, 14, 20, 23  
WinMain routine ..... 405  
WinOrgX ..... 200  
WinOrgY ..... 200  
WinX ..... 200  
WinY ..... 200  
WM\_CHANGECHAIN message ..... 296, 298  
WM\_CHAR ..... 45, 55  
WM\_COMMAND message ..... 156, 161, 176, ..... 183, 186  
WM\_CREATE message ..... 33, 116, 150, ..... 211, 230, 261, 320, 370  
WM\_DESTROY message ..... 32, 34  
WM\_DESTROYCLIPBOARD message ..... 293  
WM\_DRAWCLIPBOARD message ..... 295, ..... 296, 298

- 
- WM\_DRAWITEM message ..... 160  
WM\_ERASEBKGND message ..... 78  
WM\_HSCROLL message ..... 249, 250, 251,  
..... 254, 262  
WM\_INITDIALOG ..... 186  
WM\_INITMENU message ..... 157, 277  
WM\_INITPOP-UP message ..... 157  
WM\_KEYDOWN message ..... 45, 251, 404  
WM\_LBUTTONDOWNDBLCLK message ..... 62  
WM\_LBUTTONDOWN message ..... 32  
WM\_LBUTTONUP message ..... 62  
WM\_MDIACTIVE message ..... 411, 414  
WM\_MDICASCADE message ..... 409, 414, 427  
WM\_MDICREATE message ..... 411, 414, 426  
WM\_MDIDESTROY ..... 414  
WM\_MDIGETACTIVE ..... 414  
WM\_MDIICONARRANGE message ..... 409,  
..... 414, 427  
WM\_MDIMAXIMIZE message ..... 411, 414  
WM\_MDINEXT message ..... 411, 414  
WM\_MDIRESTORE message ..... 411, 414  
WM\_MDISETMENU message ..... 414, 428  
WM\_MDTILE message ..... 414, 427  
WM\_MEASUREITEM message ..... 160  
WM\_MOUSEMOVE message ..... 55, 57  
WM\_NC ..... 55  
WM\_NCLBUTTONDOWNDBLCLK message ..... 56  
WM\_PAINT message ..... 76, 77, 127, 252  
WM\_QUIT message ..... 31, 35  
WM\_RBUTTONDOWN message ..... 62  
WM\_RENDERALLFORMATS message ..... 293  
WM\_RENDERFORMAT message ..... 293  
WM\_SETFOCUS message ..... 405  
WM\_SIZE message ..... 116, 225  
WM\_SYSCHAR ..... 45  
WM\_SYSCOMMAND message ..... 404  
WM\_SYSKEYDOWN message ..... 44, 45  
WM\_SYSKEYUP message ..... 44  
WM\_TILE message ..... 409  
WM\_TIMER message ..... 63, 65  
WM\_VSCROLL message ..... 249, 250, 251,  
..... 254  
WM\_WININICHANGE message ..... 323  
WNDCLASS ..... 22, 23  
Word break ..... 83  
wParam parameter ..... 30, 44, 45, 55, 64,  
..... 156, 293  
WritePrivateProfileString ..... 326  
WriteProfileString ..... 324, 326  
Writing to a file ..... 307  
WS\_CHILD style parameter ..... 26, 411, 426  
WS\_CLIPCHILDREN style parameter ..... 405  
..... 407, 426  
WS\_HSCROLL parameter ..... 247, 406  
WS\_OVERLAPPED style ..... 25  
WS\_POPUP style ..... 26  
WS\_SYSMENU, style parameters ..... 411  
WS\_VSCROLL ..... 247, 406, 426  
wsprintf function ..... 54  
wUsage parameter ..... 235  
x, MDICREATESTRUCT ..... 410  
xAmount parameter ..... 253  
y, MDICREATESTRUCT ..... 410  
yAmount parameter ..... 253





# Abacus pc catalog

**Order Toll Free 1-800-451-4319**

**5370 52nd Street SE • Grand Rapids, MI 49512**  
**Phone: (616) 698-0330 • Fax: (616) 698-0325**

## Developer's Series Books

Developers Series books are for the professional software developer that requires in-depth technical information and programming techniques.

**PC System Programming** is a literal encyclopedia of technical and programming knowledge. Features parallel working examples in Pascal, C, ML and BASIC. Explains memory layout, DOS operations and interrupts from ML and high level languages, using extended, expanded memory, device drivers, hard disks, PC ports, mouse driver programming, fundamentals of BIOS, graphics and sound programming, TSR programs and complete appendices. 930 page book and two disks with over 1 MB of source code.

ISBN 1-55755-036-0. \$59.95  
Canada: 52444 \$74.95

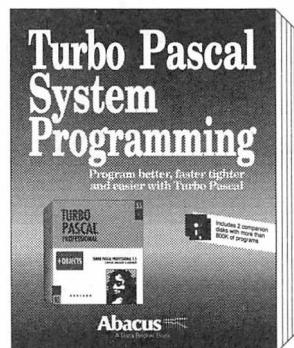
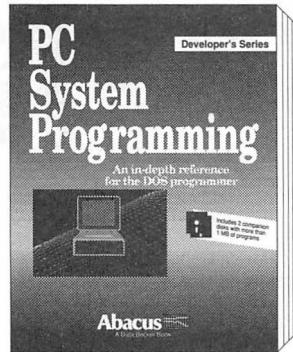
What readers are saying about  
**PC System Programming**:

"The day I saw this book I bought it" F.R.

"Excellent work" G.O.

"This book is such a good reference and so complete and in-depth, I keep it next to my system reference manuals and my compiler manuals" J.K.

"Best written PC language book I've seen" AVR, Tulsa, OK.



**Turbo Pascal System Programming** gives you the *know how* to write better, faster, tighter and easier code using the popular Turbo Pascal system. Author Michael Tischer has written this book for other pros who need to understand the ins and outs of using Turbo for their programming projects.

**Turbo Pascal System Programming** details dozens of important subjects including writing Terminate and Stay Resident (TSR) programs using Turbo; building windows and menus; using expanded and extended memory; writing multi-tasking programs; understanding Turbo's memory management;

combining assembly language with Turbo programs; using interrupts for system programming tasks; accessing the hardware directly and more.

750 page book includes two disks with more than 800K of source code.  
ISBN 1-55755-124-3. \$44.95 Available August.  
Canada: 53910 \$64.95

To order direct call Toll Free 1-800-451-4319

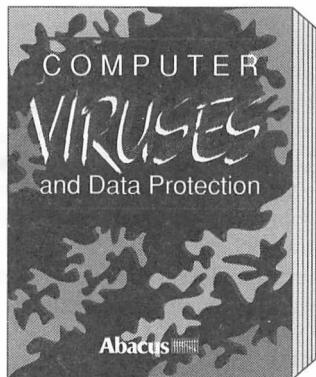
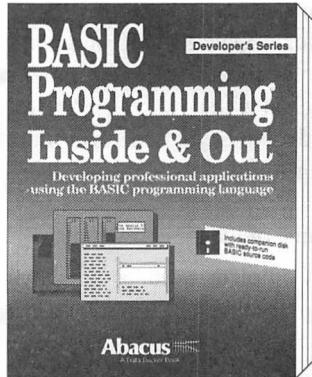
In US and Canada add \$5.00 shipping and handling. Foreign orders add \$13.00 per item.  
Michigan residents add 4% sales tax.

## Developer's Series Books

**BASIC Programming: Inside & Out**  
The standard reference for all GW-BASIC, BASIC, QuickBASIC and TurboBASIC programmers who want to be better programmers. Includes demo programs and routines you can easily adapt to your own programs. Describes sound and graphics, programming business presentation graphics, window management in PC-BASIC, using machine language, BASIC compilers, database management, create help screen editor, using pulldown menus, print multiple column and sideways and more. 600 page book with companion disk containing 360K of BASIC source code.

ISBN 1-55755-084-0. \$34.95

Canada: 54384 \$45.95



### Computer Viruses & Data Protection

describes the relatively new phenomenon among personal computer users, one that has the potential to destroy large amounts of data stored in PC systems. Simply put, this book explains what a virus is, how it works and what can be done to protect your PC against destruction.

ISBN 1-55755-123-5. \$19.95

Canada: 52089 \$25.95

To order direct call Toll Free 1-800-451-4319

In US and Canada add \$5.00 shipping and handling. Foreign orders add \$13.00 per item.  
Michigan residents add 4% sales tax.

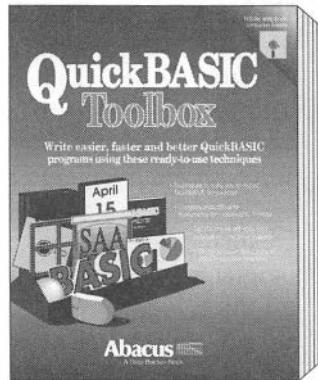
## Developer's Series Books

### QuickBASIC Toolbox

is for all QuickBASIC programmers who want professional results with minimum effort. It's packed with powerful, ready-to-use programs and routines you can use in your own programs to get professional results quickly.

Some of the topics include:

- Complete routines for SAA, interfacing mouse support, pull-down menus, windows, dialog boxes and file requestors.
- Descriptions of QuickBASIC routines.
- A BASIC Scanner program for printing completed project listings and more.

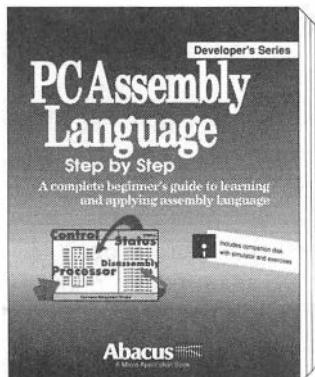


This book/disk combination will teach you how to write even better QuickBASIC code.

330 page book with companion disk.

ISBN 1-55755-104-9 \$34.95

Canada: 57911 \$45.95



### Assembly Language Step by Step

For lightning execution speed, no computer language beats assembly language. This book teaches you PC assembly and machine language the right way - one step at a time. The companion diskette contains a unique simulator which shows you how each instruction functions as the PC executes it. Includes companion diskette containing assembly language simulator.

ISBN 1-55755-096-4. \$34.95

Canada: 53927 \$45.95

To order direct call Toll Free 1-800-451-4319

In US and Canada add \$5.00 shipping and handling. Foreign orders add \$13.00 per item.  
Michigan residents add 4% sales tax.

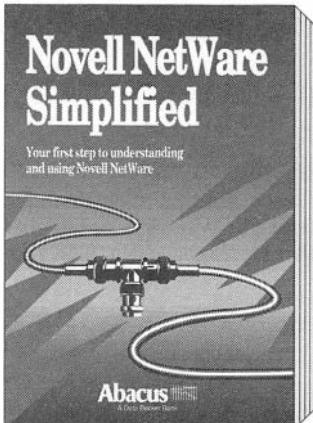
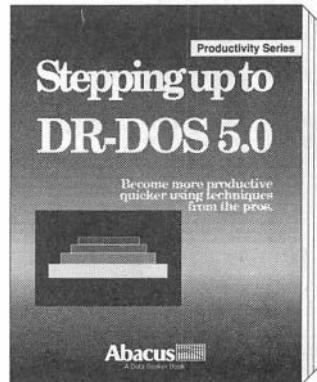
### Stepping up to DR DOS 5.0

DR DOS 5.0 is a new alternative operating system to MS-DOS. Its many new features overcome some of the limitations that users find in MS-DOS.

This fast paced guide shows you the most important features of DR DOS 5.0. It presents practical examples to get you going quickly with DR DOS 5.0. It takes you step-by-step through the world of DR DOS 5.0. You'll find clear explanations and many "hands on" examples on using DR DOS. Learn the information you'll need to become more productive with DR DOS. 210 pages.

ISBN 1-55755-106-5. \$14.95

Canada: 57913 \$19.95



### Novell NetWare Simplified

answers many questions about file servers and workstations using the most popular LAN system. **Novell NetWare Simplified** is your first step to understanding and using Novell NetWare more effectively.

Some of the topics include:

- Installing extra printers and PC workstations.
- Memory requirements for each user.
- Sending messages through your systems.
- Developing a user-friendly menu system and more.

ISBN 1-55755-105-7. \$24.95

Canada: 57910 \$3395

To order direct call Toll Free 1-800-451-4319

In US and Canada add \$5.00 shipping and handling. Foreign orders add \$13.00 per item.

Michigan residents add 4% sales tax.



## Windows System Programming

Learn to program Windows applications faster and easier.

Programming applications for Windows is said to be complex. This in-depth guide shows you how to speed up your Windows programming projects.

This bible for Windows programmers includes the tools you need to improve your programming techniques. Contains several sample applications with exhaustive documentation and background information, saving you time and effort in developing your own applications. Numerous programming examples in the popular C language lead you step-by-step through the creation process.

Topics include:

- Introduction to the Windows programming
- Using GDI for graphic and text output
- Bitmaps and the color palette manager
- Managing both static and dynamic dialog boxes
- The MDI (Multiple Document Interface)
- Data exchange between applications using DDE
- Parallel and serial interface access
- Windows memory management
- Fine tuning Windows applications
- The DLL (Dynamic Link Libraries) concept

Includes a companion diskette containing both source codes and EXE files of sample applications. The source codes can be compiled using Microsoft C Version 5.1 and higher (not included with this package) and the Microsoft Software Development Kit (SDK - not included with this package).

### **Windows System Programming.**

Authors: Peter Wilken, Dirk Honekamp

Item #B116. ISBN 1-55755-116-2.

Suggested retail price \$39.95 with companion disk. Available August.

To order direct call Toll Free 1-800-451-4319

In US and Canada add \$5.00 shipping and handling. Foreign orders add \$13.00 per item.  
Michigan residents add 4% sales tax.

## Windows Software



# New BeckerTools 2.0 *PLUS* for Windows: Indispensable utilities for every Windows user

If you're a Windows user, you'll appreciate **BeckerTools Version 2** for Windows. **BeckerTools** will dramatically speed-up your file and data management chores and increase your productivity. Where Windows' File Manager functions end, **BeckerTools** features begin by giving you powerful, yet flexible features you need to get the job done quickly. **BeckerTools** has the same consistent user interface found in Windows so there's no need to learn 'foreign' commands or functions. Manipulating your disks and files are as easy as pointing and clicking with the mouse. You'll breeze through flexible file and data functions and features that are friendly enough for a PC novice yet powerful enough for the advanced user. You won't find yourself 'dropping out' of Windows to perform powerful, essential DOS functions like undeleting a file or waiting for a diskette to be formatted. **BeckerTools** has the enhanced applications available at the click of a mouse button. Item #S110 ISBN 1-55755-110-3. With 3 1/2" and 5 1/4" diskettes. Suggested retail price \$129.95.

### Now includes—



**BeckerTools Compress**  
Defragments and optimizes disk performance.  
Optimizes your hard disk performance  
by defragmenting your files and/or disk.  
Multiple compress option lets you choose  
the type of optimization.

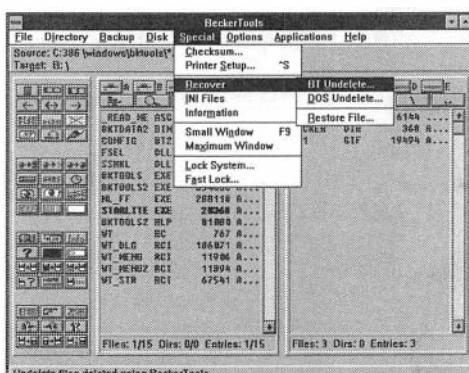


**BeckerTools Recover**  
Rescues files and disks.  
Checks and repairs common file and disk  
problems - corrupted FAT's, improperly chained  
clusters, bad files, etc. Could be a "Life-saver."



**BeckerTools Backup**  
Fast, convenient file backup and restore.  
File and disk backup/restore at the click of  
a button. Multiple select options, optional data  
compression, password protection and more.

**BeckerTools Version 2 Plus** is as easy as pointing and clicking with the mouse. **BeckerTools** takes advantage of Windows' multitasking capabilities and **BeckerTools** keeps you informed of its progress as it works.



Here are some of the things that you can easily do with **BeckerTools Version 2**:

- Launch applications - directly from **BeckerTools**
- Associate files - logically connects applications with their file types
- Backup (pack files) hard disk - saves 50% to 80% disk space - with password protection
- User levels - 3 levels for beginning, intermediate and advanced users
- Undelete - recover deleted files
- Undelete directories - recover directories in addition to individual files
- Delete files - single, groups of files or directories, including read-only files
- Duplicate diskettes - read diskette once, make multiple copies
- Edit text - built-in editor with search and replace
- Format diskettes - in any capacity supported by your drive and disk type
- Compare diskettes - in a single pass
- Wipe diskette - for maximum security
- Screen blanker - prevent CRT "burn in"
- File grouping - perform operations on files as a group
- Create a bootable system diskette
- Reliability checking - check for physical errors on disk media
- Edit files - new hex editor to edit virtually any type of file
- Dozens of other indispensable features

To order direct call Toll Free 1-800-451-4319

In US and Canada add \$5.00 shipping and handling. Foreign orders add \$13.00 per item.

Michigan residents add 4% sales tax.

## Book/companion diskette packages:

- Save hours of typing in source listings from the book.
- Provide ready-to-compile source code listings and multiple examples; help avoid printing and typing mistakes.
- This diskette contains the C source code and programs presented in the text along with many examples to make you a Windows programmer now.

If you bought this book without the diskette, call us today to order our economical companion diskette and save yourself valuable time.



5370 52nd Street SE • Grand Rapids, MI 49512  
Call 1-800-451-4319

## Intro to Windows Programming Companion Diskette

The companion diskette contains all of the C sample programs in this book; both in source code format and in executable form. All programs written for Microsoft C 6.00 and the Windows Software Development Kit (SDK); not included.

The following directories contain the examples listed in the book.

WINPRGDE	: Main directory for all programs
FIRSTAPP	: Your first Windows application
KBDMOUSE	: Accessing the mouse and keyboard for input
OUTPUT	: Basic screen output using text, graphics, tools
RESOURCE	: Applications using resources
MENU	: Accesses menus using resources
FRSTMENU	: Simple menu from resources
ICONS	: Accesses icons from resources
MAPPING	: Mapping modes application
BITMAPS	: Two bitmap applications
SCROLBAR	: Scroll bar application
CLIPBOARD	: Clipboard access applications
FILEMGMT	: File access applications
FILE	: Simple database
FINIT	: Create .INI file
DLL	: Dynamic Link Library applications
DLL_RC	: Application and DLL using resources
DLL_FUNC	: Application and DLL using functions
HELPAPP	: Creating a help system for an application
MDI	: The Multiple Document Interface
BATCHES	: Batch files used to compile and link applications
EXECS	: Executable versions of all files on this diskette

- The companion diskette allows you to start using the source codes and programs presented in the book immediately.