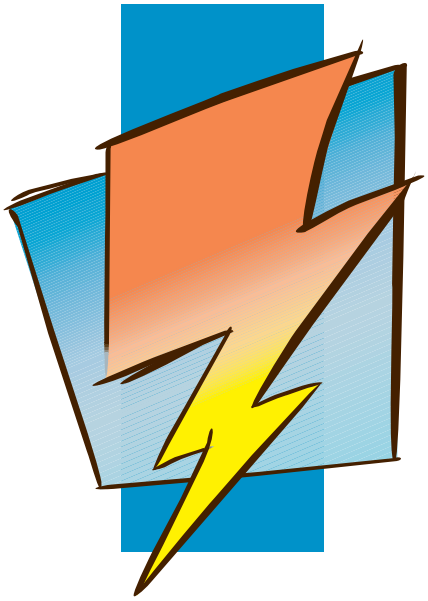
Open Watcom C/C++

Programmer's Guide



Version 2.0

Open Watcom

Notice of Copyright

For more information please visit http://www.openwatcom.org/

Portions of this manual are reprinted with permission from Tenberry Software, Inc.

Preface

The ***Open Watcom C/C++ Programmer's Guide*** includes the following major components:

- DOS Programming Guide
- The DOS/4GW DOS Extender
- Windows 3.x Programming Guide
- Windows NT Programming Guide
- OS/2 Programming Guide
- Novell NLM Programming Guide
- Mixed Language Programming
- Common Problems

Acknowledgements

This book was produced with the Open Watcom GML electronic publishing system, a software tool developed by WATCOM. In this system, writers use an ASCII text editor to create source files containing text annotated with tags. These tags label the structural elements of the document, such as chapters, sections, paragraphs, and lists. The Open Watcom GML software, which runs on a variety of operating systems, interprets the tags to format the text into a form such as you see here. Writers can produce output for a variety of printers, including laser printers, using separately specified layout directives for such things as font selection, column width and height, number of columns, etc. The result is type-set quality copy containing integrated text and graphics.

Many users have provided valuable feedback on earlier versions of the Open Watcom C/C++ compilers and related tools. Their comments were greatly appreciated. If you find problems in the documentation or have some good suggestions, we would like to hear from you.

July, 1997.

Trademarks Used in this Manual

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

List of Figures

1 Open Watcom C/C++ Application Development

This document contains guides to application development for several environments including 16-bit DOS, 32-bit extended DOS, Windows 3.x, 32-bit extended Windows 3.x, Windows NT/2000/XP, Win9x, OS/2, and Novell NLMs. It also describes mixed language (C, FORTRAN) application development. It concludes with a chapter on some general questions and the answers to them.

This document covers the following topics:

- DOS Programming Guide

  Creating 16-bit DOS Applications
  Creating 32-bit Phar Lap 386\|DOS-Extender Applications
  Creating 32-bit DOS/4GW Applications
  32-bit Extended DOS Application Development

- The DOS/4GW DOS Extender

  The Tenberry Software DOS/4GW DOS Extender
  Linear Executables
  Configuring DOS/4GW
  VMM
  Interrupt 21H Functions
  Interrupt 31H DPMI Functions
  Utilities
  Error Messages
  DOS/4GW Commonly Asked Questions

- Windows 3.x Programming Guide

  Creating 16-bit Windows 3.x Applications
  Porting Non-GUI Applications to 16-bit Windows 3.x
  Creating 32-bit Windows 3.x Applications
  Porting Non-GUI Applications to 32-bit Windows 3.x
  The Open Watcom 32-bit Windows Extender
  Windows 3.x 32-bit Programming Overview
  Windows 32-Bit Dynamic Link Libraries
  Interfacing Visual Basic and Open Watcom C/C++ DLLs
  WIN386 Library Functions and Macros
  32-bit Extended Windows Application Development
  Special Variables for Windows Programming
  Definitions of Windows Terms
  Special Windows API Functions

- Windows NT Programming Guide

Windows NT Programming Overview
Creating Windows NT GUI Applications
Porting Non-GUI Applications to Windows NT GUI
Windows NT Multi-threaded Applications
Windows NT Dynamic Link Libraries

- OS/2 Programming Guide

Creating 16-bit OS/2 1.x Applications
Creating 32-bit OS/2 Applications
OS/2 Multi-threaded Applications
OS/2 Dynamic Link Libraries
Programming for OS/2 Presentation Manager

- Novell NLM Programming Guide

Creating NetWare 386 NLM Applications

- Mixed Language Programming

Inter-Language calls: C and FORTRAN

- Common Problems

Commonly Asked Questions and Answers

DOS Programming Guide

2 Creating 16-bit DOS Applications

This chapter describes how to compile and link 16-bit DOS applications simply and quickly.

We will illustrate the steps to creating 16-bit DOS applications by taking a small sample application and showing you how to compile, link, run and debug it.

2.1 The Sample Application

To demonstrate the creation of 16-bit DOS applications using command-line oriented tools, we introduce a simple sample program. For our example, we are going to use the famous "hello" program.

```
#include <stdio.h>

void main()
{
    printf( "Hello world\n" );
}
```

The C++ version of this program follows:

```
#include <iostream.h>

void main()
{
    cout << "Hello world" << endl;
}
```

The goal of this program is to display the message "Hello world" on the screen. The C version uses the C library `printf` routine to accomplish this task. The C++ version uses the "iostream" library to accomplish this task. We will take you through the steps necessary to produce this result.

2.2 Building and Running the Sample DOS Application

To compile and link our example program which is stored in the file `hello.c`, enter the following command:

```
C>wcl -l=dos hello.c
```

The typical messages that appear on the screen are shown in the following illustration.

```
C>wcl -l=dos  hello.c
Open Watcom C/C++16 Compile and Link Utility
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1988-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
        wcc hello.c
Open Watcom C16 Optimizing Compiler
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1984-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
hello.c: 6 lines, included 155, 0 warnings, 0 errors
Code size: 17

Open Watcom Linker
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1985-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
loading object files
searching libraries
creating a DOS executable
```

Provided that no errors were encountered during the compile or link phases, the "hello" program may now be run.

```
C>hello
Hello world
```

If you examine the current directory, you will find that two files have been created. These are `hello.obj` (the result of compiling `hello.c`) and `hello.exe` (the result of linking `hello.obj` with the appropriate Open Watcom C/C++ libraries). It is `hello.exe` that is run by DOS when you enter the "hello" command.

2.3 Debugging the Sample DOS Application

Let us assume that you wish to debug your application in order to locate an error in programming. In the previous section, the "hello" program was compiled with default compile and link options. When debugging an application, it is useful to refer to the symbolic names of routines and variables. It is also convenient to debug at the source line level rather than the machine language level. To do this, we must direct both the compiler and linker to include additional debugging information in the object and executable files. Using the ***WCL*** command, this is fairly straightforward. ***WCL*** recognizes the Open Watcom C/C++ compiler "debug" options and will create the appropriate debug directives for the Open Watcom Linker.

For example, to compile and link the "hello" program with debugging information, the following command may be issued.

```
C>wcl -l=dos -d2 hello.c
```

The typical messages that appear on the screen are shown in the following illustration.

```
C>wcl -l=dos  -d2 hello.c
Open Watcom C/C++16 Compile and Link Utility
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1988-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
        wcc hello.c  -d2
Open Watcom C16 Optimizing Compiler
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1984-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
hello.c: 6 lines, included 155, 0 warnings, 0 errors
Code size: 23

Open Watcom Linker
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1985-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
loading object files
searching libraries
creating a DOS executable
```

The "d2" option requests the maximum amount of debugging information that can be provided by the Open Watcom C/C++ compiler. ***WCL*** will make sure that this debugging information is included in the executable file that is produced by the linker.

The "Code size" value is larger than in the previous example since selection of the "d2" option results in fewer code optimizations by default. You can request more optimization by specifying the appropriate options. However, you do so at the risk of making it more difficult for yourself to determine the relationship between the object code and the original source language code.

To request the Open Watcom Debugger to assist in debugging the application, the following command may be issued.

```
C>wd hello
```

It would be too ambitious to describe the debugger in this introductory chapter so we refer you to the book entitled ***Open Watcom Debugger User's Guide.***

3 Creating 32-bit Phar Lap 386|DOS-Extender Applications

This chapter describes how to compile and link 32-bit Phar Lap 386|DOS-Extender applications simply and quickly.

We will illustrate the steps to creating 32-bit Phar Lap 386|DOS-Extender applications by taking a small sample application and showing you how to compile, link, run and debug it.

3.1 The Sample Application

To demonstrate the creation of 32-bit Phar Lap 386|DOS-Extender applications using command-line oriented tools, we introduce a simple sample program. For our example, we are going to use the famous "hello" program.

```
#include <stdio.h>

void main()
{
    printf( "Hello world\n" );
}
```

The C++ version of this program follows:

```
#include <iostream.h>

void main()
{
    cout << "Hello world" << endl;
}
```

The goal of this program is to display the message "Hello world" on the screen. The C version uses the C library `printf` routine to accomplish this task. The C++ version uses the "iostream" library to accomplish this task. We will take you through the steps necessary to produce this result.

3.2 Building and Running the Sample 386|DOS-Extender Application

To compile and link our example program which is stored in the file `hello.c`, enter the following command:

```
C>wcl386 -l=pharlap hello.c
```

The typical messages that appear on the screen are shown in the following illustration.

```
C>wcl386 -l=pharlap  hello.c
Open Watcom C/C++32 Compile and Link Utility
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1988-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
        wcc386 hello.c
Open Watcom C32 Optimizing Compiler
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1984-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
hello.c: 6 lines, included 174, 0 warnings, 0 errors
Code size: 24

Open Watcom Linker
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1985-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
loading object files
searching libraries
creating a Phar Lap simple executable
```

Provided that no errors were encountered during the compile or link phases, the "hello" program may now be run.

```
C>run386 hello
Hello world
```

If you examine the current directory, you will find that two files have been created. These are `hello.obj` (the result of compiling `hello.c`) and `hello.exp` (the result of linking `hello.obj` with the appropriate Open Watcom C/C++ libraries). It is `hello.exp` that is run by DOS when you enter the "run386 hello" command.

3.3 Debugging the Sample 386|DOS-Extender Application

Let us assume that you wish to debug your application in order to locate an error in programming. In the previous section, the "hello" program was compiled with default compile and link options. When debugging an application, it is useful to refer to the symbolic names of routines and variables. It is also convenient to debug at the source line level rather than the machine language level. To do this, we must direct both the compiler and linker to include additional debugging information in the object and executable files. Using the ***WCL386*** command, this is fairly straightforward. ***WCL386*** recognizes the Open Watcom C/C++ compiler "debug" options and will create the appropriate debug directives for the Open Watcom Linker.

For example, to compile and link the "hello" program with debugging information, the following command may be issued.

```
C>wcl386 -l=pharlap -d2 hello.c
```

The typical messages that appear on the screen are shown in the following illustration.

```
C>wcl386 -l=pharlap  -d2 hello.c
Open Watcom C/C++32 Compile and Link Utility
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1988-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
        wcc386 hello.c  -d2
Open Watcom C32 Optimizing Compiler
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1984-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
hello.c: 6 lines, included 174, 0 warnings, 0 errors
Code size: 45

Open Watcom Linker
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1985-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
loading object files
searching libraries
creating a Phar Lap simple executable
```

The "d2" option requests the maximum amount of debugging information that can be provided by the Open Watcom C/C++ compiler. ***WCL386*** will make sure that this debugging information is included in the executable file that is produced by the linker.

The "Code size" value is larger than in the previous example since selection of the "d2" option results in fewer code optimizations by default. You can request more optimization by specifying the appropriate options. However, you do so at the risk of making it more difficult for yourself to determine the relationship between the object code and the original source language code.

To request the Open Watcom Debugger to assist in debugging the application, the following command may be issued.

```
C>wd /trap=pls hello
```

It would be too ambitious to describe the debugger in this introductory chapter so we refer you to the book entitled ***Open Watcom Debugger User's Guide.***

4 Creating 32-bit DOS/4GW Applications

This chapter describes how to compile and link 32-bit DOS/4GW applications simply and quickly.

We will illustrate the steps to creating 32-bit DOS/4GW applications by taking a small sample application and showing you how to compile, link, run and debug it.

4.1 The Sample Application

To demonstrate the creation of 32-bit DOS/4GW applications using command-line oriented tools, we introduce a simple sample program. For our example, we are going to use the famous "hello" program.

```
#include <stdio.h>

void main()
{
    printf( "Hello world\n" );
}
```

The C++ version of this program follows:

```
#include <iostream.h>

void main()
{
    cout << "Hello world" << endl;
}
```

The goal of this program is to display the message "Hello world" on the screen. The C version uses the C library `printf` routine to accomplish this task. The C++ version uses the "iostream" library to accomplish this task. We will take you through the steps necessary to produce this result.

4.2 Building and Running the Sample DOS/4GW Application

To compile and link our example program which is stored in the file `hello.c`, enter the following command:

```
C>wcl386 -l=dos4g hello.c
```

The typical messages that appear on the screen are shown in the following illustration.

```
C>wcl386 -l=dos4g  hello.c
Open Watcom C/C++32 Compile and Link Utility
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1988-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
        wcc386 hello.c
Open Watcom C32 Optimizing Compiler
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1984-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
hello.c: 6 lines, included 174, 0 warnings, 0 errors
Code size: 24

Open Watcom Linker
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1985-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
loading object files
searching libraries
creating a DOS/4G executable
```

Provided that no errors were encountered during the compile or link phases, the "hello" program may now be run.

```
C>hello
Hello world
```

If you examine the current directory, you will find that two files have been created. These are `hello.obj` (the result of compiling `hello.c`) and `hello.exe` (the result of linking `hello.obj` with the appropriate Open Watcom C/C++ libraries). It is `hello.exe` that is run by DOS when you enter the "hello" command.

4.3 Debugging the Sample DOS/4GW Application

Let us assume that you wish to debug your application in order to locate an error in programming. In the previous section, the "hello" program was compiled with default compile and link options. When debugging an application, it is useful to refer to the symbolic names of routines and variables. It is also convenient to debug at the source line level rather than the machine language level. To do this, we must direct both the compiler and linker to include additional debugging information in the object and executable files. Using the ***WCL386*** command, this is fairly straightforward. ***WCL386*** recognizes the Open Watcom C/C++ compiler "debug" options and will create the appropriate debug directives for the Open Watcom Linker.

For example, to compile and link the "hello" program with debugging information, the following command may be issued.

```
C>wcl386 -l=dos4g -d2 hello.c
```

The typical messages that appear on the screen are shown in the following illustration.

```
C>wcl386 -l=dos4g  -d2 hello.c
Open Watcom C/C++32 Compile and Link Utility
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1988-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
        wcc386 hello.c  -d2
Open Watcom C32 Optimizing Compiler
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1984-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
hello.c: 6 lines, included 174, 0 warnings, 0 errors
Code size: 45

Open Watcom Linker
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1985-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
loading object files
searching libraries
creating a DOS/4G executable
```

The "d2" option requests the maximum amount of debugging information that can be provided by the Open Watcom C/C++ compiler. ***WCL386*** will make sure that this debugging information is included in the executable file that is produced by the linker.

The "Code size" value is larger than in the previous example since selection of the "d2" option results in fewer code optimizations by default. You can request more optimization by specifying the appropriate options. However, you do so at the risk of making it more difficult for yourself to determine the relationship between the object code and the original source language code.

To request the Open Watcom Debugger to assist in debugging the application, the following command may be issued.

```
C>wd /trap=rsi hello
```

It would be too ambitious to describe the debugger in this introductory chapter so we refer you to the book entitled ***Open Watcom Debugger User's Guide.***

5 32-bit Extended DOS Application Development

5.1 Introduction

The purpose of this chapter is to anticipate common programming questions for 32-bit extended DOS application development. Note that these programming solutions may be DOS-extender specific and therefore may not work for other DOS extenders.

The following topics are discussed in this chapter:

- How can I write directly to video memory using a DOS extender?
- How do I get information about free memory in the 32-bit environment?
- How do I access the first megabyte in the extended DOS environment?
- How do I spawn a protected-mode application?
- How can I use the mouse interrupt (0x33) with DOS/4GW?
- How do I simulate a real-mode interrupt with DOS/4GW?
- How do you install a bi-modal interrupt handler with DOS/4GW?

Please refer to the ***DOS Protected-Mode Interface (DPMI) Specification*** for information on DPMI services. In the past, the DPMI specification could be obtained free of charge by contacting Intel Literature JP26 at 800-548-4725 or by writing to the address below. We have been advised that the DPMI specification is no longer available in printed form.

Intel Literature JP26
3065 Bowers Avenue
P.O. Box 58065
Santa Clara, California
U.S.A. 95051-8065

However, the DPMI 1.0 specification can be obtained from the Intel ftp site. Here is the URL.

```
ftp://ftp.intel.com/pub/IAL/software_ specs/dpmiv1.zip
```

This ZIP file contains a Postscript version of the DPMI 1.0 specification.

5.2 How can I write directly to video memory using a DOS extender?

Many programmers require access to video RAM in order to directly manipulate data on the screen. Under DOS, it was standard practice to use a far pointer, with the segment part of the far pointer set to the screen segment. Under DOS extenders, this practice is not so standard. Each DOS extender provides its own method for accessing video memory.

5.2.1 Writing to Video Memory under Tenberry Software DOS/4GW

Under DOS/4GW, the first megabyte of physical memory is mapped as a shared linear address space. This allows your application to access video RAM using a near pointer set to the screen's linear address. The following program demonstrates this method.

```
/*
    SCREEN.C - This example shows how to write directly
    to screen memory under the DOS/4GW dos-extender.

    Compile & Link: wcl386 -l=dos4g SCREEN
 */
#include <stdio.h>
#include <dos.h>

/*
  Under DOS/4GW, the first megabyte of physical memory
  (real-mode memory) is mapped as a shared linear address
  space. This allows your application to access video RAM
  using its linear address.  The DOS segment:offset of
  B800:0000 corresponds to a linear address of B8000.
 */
#define SCREEN_AREA 0xb800
#define SCREEN_LIN_ADDR ((SCREEN_AREA) << 4)
#define SCREEN_SIZE 80*25
void main()
{
    char       *ptr;
    int         i;

    /* Set the pointer to the screen's linear address */
    ptr = (char *)SCREEN_LIN_ADDR;
    for( i = 0; i < SCREEN_SIZE - 1; i++ ) {
        *ptr = '*';
        ptr += 2 * sizeof( char );
    }
}
```

Please refer to the chapter entitled "Linear Executables" on page 43 for more information on how DOS/4GW maps the first megabyte.

5.2.2 Writing to Video Memory under the Phar Lap 386|DOS-Extender

The Phar Lap DOS extender provides screen access through the special segment selector 0x1C. This allows far pointer access to video RAM from a 32-bit program. The following example illustrates this technique.

```
/*
    SCREENPL.C - This example shows how to write directly
    to screen memory under the Phar Lap DOS extender.

    Compile & Link: wcl386 -l=pharlap SCREENPL
 */
#include <stdio.h>
#include <dos.h>

/*
    Phar Lap allows access to screen memory through a
    special selector.  Refer to "Hardware Access" in
    Phar Lap's documentation for details.
 */
#define PL_ SCREEN_ SELECTOR 0x1c
#define SCREEN_ SIZE 80*25
void main()
{
    /* Need a far pointer to use the screen selector */
    char far   *ptr;
    int         i;

    /* Make a far pointer to screen memory */
    ptr = MK_ FP( PL_ SCREEN_ SELECTOR, 0 );
    for( i = 0; i < SCREEN_ SIZE - 1; i++ ) {
        *ptr = '*';
        ptr += 2 * sizeof( char );
    }
}
```

It is also possible to map screen memory into your near memory using Phar Lap system calls. Please refer to the chapter entitled "386|DOS-Extender System Calls" in Phar Lap's ***386|DOS-Extender Reference Manual*** for details.

5.3 How do I get information about free memory in the 32-bit environment?

Under a virtual memory system, programmers are often interested in the amount of physical memory they can allocate. Information about the amount of free memory that is available is always provided under a DPMI host, however, the manner in which this information is provided may differ under various environments. Keep in mind that in a multi-tasking environment, the information returned to your task from the DPMI host can easily become obsolete if other tasks allocate memory independently of your task.

5.3.1 Getting Free Memory Information under DOS/4GW

DOS/4GW provides a DPMI interface through interrupt 0x31. This allows you to use DPMI service 0x0500 to get free memory information. The following program illustrates this procedure.

```
/*
    MEMORY.C - This example shows how to get information
    about free memory using DPMI call 0500h under DOS/4GW.
    Note that only the first field of the structure is
    guaranteed to contain a valid value; any field that
    is not returned by DOS/4GW is set to -1 (0FFFFFFFFh).

    Compile & Link: wcl386 -l=dos4g memory
 */
#include <i86.h>
#include <dos.h>
#include <stdio.h>

#define DPMI_INT        0x31

struct meminfo {
    unsigned LargestBlockAvail;
    unsigned MaxUnlockedPage;
    unsigned LargestLockablePage;
    unsigned LinAddrSpace;
    unsigned NumFreePagesAvail;
    unsigned NumPhysicalPagesFree;
    unsigned TotalPhysicalPages;
    unsigned FreeLinAddrSpace;
    unsigned SizeOfPageFile;
    unsigned Reserved[3];
} MemInfo;

void main()
{
    union REGS regs;
    struct SREGS sregs;

    regs.x.eax = 0x00000500;
    memset( &sregs, 0, sizeof(sregs) );
    sregs.es = FP_SEG( &MemInfo );
    regs.x.edi = FP_OFF( &MemInfo );
```

```
    int386x( DPMI_INT, &regs, &regs, &sregs );
    printf( "Largest available block (in bytes): %lu\n",
            MemInfo.LargestBlockAvail );
    printf( "Maximum unlocked page allocation: %lu\n",
            MemInfo.MaxUnlockedPage );
    printf( "Pages that can be allocated and locked: "
            "%lu\n", MemInfo.LargestLockablePage );
    printf( "Total linear address space including "
            "allocated pages: %lu\n",
            MemInfo.LinAddrSpace );
    printf( "Number of free pages available: %lu\n",
             MemInfo.NumFreePagesAvail );
    printf( "Number of physical pages not in use: %lu\n",
             MemInfo.NumPhysicalPagesFree );
    printf( "Total physical pages managed by host: %lu\n",
             MemInfo.TotalPhysicalPages );
    printf( "Free linear address space (pages): %lu\n",
             MemInfo.FreeLinAddrSpace );
    printf( "Size of paging/file partition (pages): %lu\n",
             MemInfo.SizeOfPageFile );
}
```

Please refer to the chapter entitled "Interrupt 31H DPMI Functions" on page 61 for more information on DPMI services.

5.3.2 Getting Free Memory Information under the Phar Lap 386|DOS-Extender

Phar Lap provides memory statistics through 386|DOS-Extender System Call 0x2520. The following example illustrates how to use this system call from a 32-bit program.

```
/*
    MEMPLS40.C - This is an example of how to get the
    amount of physical memory present under Phar Lap
    386|DOS-Extender v4.0.

    Compile & Link: wcl386 -l=pharlap MEMPLS40
 */
#include <dos.h>
#include <stdio.h>

typedef struct {
    unsigned data[25];
} pharlap_mem_status;

/* Names suggested in Phar Lap documentation */
#define APHYSPG     5
#define SYSPHYSPG   7
#define NFREEPG     21

unsigned long memavail( void )
{
    pharlap_mem_status status;
    union REGS regs;
    unsigned long amount;
```

```
    regs.h.ah = 0x25;
    regs.h.al = 0x20;
    regs.h.bl = 0;
    regs.x.edx = (unsigned int) &status;
    intdos( &regs, &regs );
    /* equation is given in description for nfreepg */
    amount = status.data[ APHYSPG ];
    amount += status.data[ SYSPHYSPG ];
    amount += status.data[ NFREEPG ];
    return( amount * 4096 );
}

void main()
{
    printf( "%lu bytes of memory available\n",
            memavail() );
}
```

Please refer to the chapter entitled "386|DOS-Extender System Calls" in Phar Lap's ***386|DOS-Extender Reference Manual*** for more information on 386|DOS-Extender System Calls.

5.3.3 Getting Free Memory Information in the 32-bit Environment under Windows 3.x

Windows 3.x provides a DPMI host that you can access from a 32-bit program. The interface to this host is a 16-bit interface, hence there are some considerations involved when calling Windows 3.x DPMI services from 32-bit code. If a pointer to a data buffer is required to be passed in ES:DI, for example, an AllocAlias16() may be used to get a 16-bit far pointer that can be passed to Windows 3.x through these registers. Also, an int86() call should be issued rather than an int386() call. The following program demonstrates the techniques mentioned above.

```
/*
    MEMWIN.C - This example shows how to get information
    about free memory with DPMI call 0x0500 using Windows
    as a DPMI host.  Note that only the first field of the
    structure is guaranteed to contain a valid value; any
    field that is not returned by the DPMI implementation
    is set to -1 (0FFFFFFFFh).

    Compile & Link: wcl386 -l=win386 -zw memwin
    Bind: wbind -n memwin
 */
#include <windows.h>
#include <i86.h>
#include <dos.h>
#include <stdio.h>
```

```
struct meminfo {
    unsigned LargestBlockAvail;
    unsigned MaxUnlockedPage;
    unsigned LargestLockablePage;
    unsigned LinAddrSpace;
    unsigned NumFreePagesAvail;
    unsigned NumPhysicalPagesFree;
    unsigned TotalPhysicalPages;
    unsigned FreeLinAddrSpace;
    unsigned SizeOfPageFile;
    unsigned Reserved[3];
} MemInfo;

#define DPMI_INT        0x31
void main()
{
    union REGS regs;
    struct SREGS sregs;
    DWORD mi_16;

    regs.w.ax = 0x0500;
    mi_16 = AllocAlias16( &MemInfo );
    sregs.es = HIWORD( mi_16 );
    regs.x.di = LOWORD( mi_16 );

    int86x( DPMI_INT, &regs, &regs, &sregs );
    printf( "Largest available block (in bytes): %lu\n",
            MemInfo.LargestBlockAvail );
    printf( "Maximum unlocked page allocation: %lu\n",
            MemInfo.MaxUnlockedPage );
    printf( "Pages that can be allocated and locked: "
            "%lu\n", MemInfo.LargestLockablePage );
    printf( "Total linear address space including "
            "allocated pages: %lu\n",
            MemInfo.LinAddrSpace );
    printf( "Number of free pages available: %lu\n",
             MemInfo.NumFreePagesAvail );
    printf( "Number of physical pages not in use: %lu\n",
             MemInfo.NumPhysicalPagesFree );
    printf( "Total physical pages managed by host: %lu\n",
             MemInfo.TotalPhysicalPages );
    printf( "Free linear address space (pages): %lu\n",
             MemInfo.FreeLinAddrSpace );
    printf( "Size of paging/file partition (pages): %lu\n",
             MemInfo.SizeOfPageFile );
    FreeAlias16( mi_16 );
}
```

Please refer to the ***DOS Protected-Mode Interface (DPMI) Specification*** for information on DPMI services. In the past, the DPMI specification could be obtained free of charge by contacting Intel Literature JP26 at 800-548-4725 or by writing to the address below. We have been advised that the DPMI specification is no longer available in printed form.

Intel Literature JP26
3065 Bowers Avenue
P.O. Box 58065
Santa Clara, California
U.S.A. 95051-8065

However, the DPMI 1.0 specification can be obtained from the Intel ftp site. Here is the URL.

```
ftp://ftp.intel.com/pub/IAL/software_ specs/dpmiv1.zip
```

This ZIP file contains a Postscript version of the DPMI 1.0 specification.

5.4 How do I access the first megabyte in the extended DOS environment?

Many programmers require access to the first megabyte of memory in order to look at key low memory addresses. Under DOS, it was standard practice to use a far pointer, with the far pointer set to the segmented address of the memory area that was being inspected. Under DOS extenders, this practice is not so standard. Each DOS extender provides its own method for accessing the first megabyte of memory.

5.4.1 Accessing the First Megabyte under Tenberry Software DOS/4GW

Under DOS/4GW, the first megabyte of physical memory - the real memory - is mapped as a shared linear address space. This allows your application to access the first megabyte of memory using a near pointer set to the linear address. The following program demonstrates this method. This example is similar to the screen memory access example.

```
/*
    KEYSTAT.C - This example shows how to get the keyboard
    status under DOS/4GW by looking at the ROM BIOS
    keyboard status byte in low memory.

    Compile & Link: wcl386 -l=dos4g keystat
 */
#include <stdio.h>
#include <dos.h>

/*
    Under DOS, the keyboard status byte has a segmented
    address of 0x0040:0x0017.  This corresponds to a
    linear address of 0x417.
 */
#define LOW_ AREA 0x417

void main()
{
    /* Only need a near pointer in the flat model */
    char *ptr;

    /* Set pointer to linear address of the first
       status byte */
    ptr = (char *)LOW_ AREA;

    /* Caps lock state is in bit 6 */
    if( *ptr & 0x40 ) {
        puts( "Caps Lock on" );
    }
```

```
    /* Num lock state is in bit 5 */
    if( *ptr & 0x20 ) {
        puts( "Num Lock on" );
    }
    /* Scroll lock state is in bit 4 */
    if( *ptr & 0x10 ) {
        puts( "Scroll Lock on" );
    }
}
```

Please refer to the chapter entitled "Linear Executables" on page 43 for more information on how DOS/4GW maps the first megabyte.

5.4.2 Accessing the First Megabyte under the Phar Lap 386|DOS-Extender

The Phar Lap DOS extender provides access to real memory through the special segment selector 0x34. This allows far pointer access to the first megabyte from a 32-bit program. The following example illustrates this technique.

```
/*
    KEYSTAPL.C - This example shows how to get the keyboard
    status under 386|DOS-Extender by looking at the ROM
    BIOS keyboard status byte in low memory.

    Compile & Link: wcl386 -l=pharlap keystapl
 */
#include <stdio.h>
#include <dos.h>

/*
    Under DOS, the keyboard status byte has a segmented
    address of 0x0040:0x0017.  This corresponds to a
    linear address of 0x417.
 */

void main()
{
    /* We require a far pointer to use selector
       for 1st megabyte */
    char far *ptr;
    /* Set pointer to segmented address of the first
       status byte */
    ptr = MK_FP( 0x34, 0x417 );
    /* Caps lock state is in bit 6 */
    if( *ptr & 0x40 ) {
        puts( "Caps Lock on" );
    }
    /* Num lock state is in bit 5 */
    if( *ptr & 0x20 ) {
        puts( "Num Lock on" );
    }
    /* Scroll lock state is in bit 4 */
    if( *ptr & 0x10 ) {
        puts( "Scroll Lock on" );
    }
}
```

Please refer to the chapter entitled "Program Environment" in Phar Lap's ***386\|DOS-Extender Reference Manual*** for more information on segment selectors available to your program.

5.5 How do I spawn a protected-mode application?

Sometimes applications need to spawn other programs as part of their execution. In the extended DOS environment, spawning tasks is much the same as under DOS, however it should be noted that the only mode supported is P_WAIT. The P_OVERLAY mode is not supported since the DOS extender cannot be removed from memory by the application (this is also the reason why the exec() functions are unsupported). The other modes are for concurrent operating systems only.

Also, unless the application being spawned is bound or stubbed, the DOS extender must be spawned with the application and its arguments passed in the parameter list.

5.5.1 Spawning Protected-Mode Applications Under Tenberry Software DOS/4GW

In the case of DOS/4GW, some real-mode memory must be set aside at run time for spawning the DOS extender, otherwise the spawning application could potentially allocate all of system memory. The real memory can be reserved from within your program by assigning the global variable `__minreal` the number of bytes to be set aside. This variable is referenced in `<stdlib.h>`. The following two programs demonstrate how to spawn a DOS/4GW application.

```
/*
    SPWNRD4G.C - The following program demonstrates how to
    spawn another DOS/4GW application.

    Compile and link: wcl386 -l=dos4g spwnrd4g
 */
#include <process.h>
#include <stdio.h>
#include <stdlib.h>

/* DOS/4GW var for WLINK MINREAL option */
unsigned __near __minreal = 100*1024;

void main()
{
    int app2_exit_code;

    puts( "Spawning a protected-mode application..."
          "using spawnlp() with P_WAIT" );
    app2_exit_code = spawnlp( P_WAIT, "dos4gw",
                              "dos4gw", "spwndd4g", NULL );
    printf( "Application #2 returned with exit code %d\n",
                              app2_exit_code );
}
```

```
/*
    SPWNDD4G.C - Will be spawned by the SPWNRD4G program.

    Compile & Link: wcl386 -l=dos4g spwndd4g
 */
#include <stdio.h>
#include <stdlib.h>

void main()
{
    puts( "\nApplication #2 spawned\n" );
    /* Send back exit code 59 */
    exit( 59 );
}
```

5.5.2 Spawning Protected-Mode Applications Under Phar Lap 386|DOS-Extender

In the case of the Phar Lap 386|DOS-Extender, some real-mode memory must be set aside at link time for spawning the DOS extender, otherwise the spawning application will be assigned all the system memory at startup. This is done at link time by specifying the **runtime minreal** and **runtime maxreal** options, as demonstrated by the following programs.

```
/*
    SPWNRPLS.C - The following program demonstrates how to
    spawn a Phar Lap application.

    Compile & Link:
    wcl386 -l=pharlap -"runt minr=300K,maxr=400K" spwnrpls
 */
#include <process.h>
#include <stdio.h>

void main()
{
    int app2_ exit_ code;

    puts( "Spawning a protect-mode application..."
          "using spawnlp() with P_ WAIT" );
    puts( "Spawning application #2..." );
    app2_ exit_ code = spawnlp( P_ WAIT, "run386",
                               "run386", "spwndpls", NULL );

    printf( "Application #2 returned with exit code %d",
                app2_ exit_ code );
}
```

```
/*
    SPWNDPLS.C - Will be spawned by the SPWNRPLS program.

    Compile & Link: wcl386 -l=pharlap spwndpls
 */
#include <stdio.h>
#include <stdlib.h>

void main()
{
    puts( "\nApplication #2 spawned\n" );
    /* Exit with error code 59 */
    exit( 59 );
}
```

5.6 How Can I Use the Mouse Interrupt (0x33) with DOS/4GW?

Several commonly used interrupts are automatically supported in protected mode with DOS/4GW. The DOS extender handles the switch from protected mode to real mode and manages any intermediate real-mode data buffers that are required. To use a supported interrupt, set up the register information as required for the interrupt and use one of the int386() or int386x() library functions to execute the interrupt. For calls that are not supported by DOS/4GW, you can use the DPMI function, Simulate a Real-Mode Interrupt (0x0300). This process is described in the next section.

Since the mouse interrupt (0x33) is quite commonly used, DOS/4GW provides protected-mode support for the interrupt and any mouse data buffer that is required. The following example demonstrates how a programmer could use the Microsoft standard mouse interrupt (0x33) from within a DOS/4GW application.

```
/*
        mouse.c - The following program demonstrates how
        to use the mouse interrupt (0x33) with DOS/4GW.

        Compile and link: wcl386 -l=dos4g mouse
*/
#include <stdio.h>
#include <dos.h>
#include <i86.h>

/* Data touched at mouse callback time -- they are
   in a structure to simplify calculating the size
   of the region to lock.
*/
struct callback_data {
    int             right_button;
    int             mouse_event;
    unsigned short  mouse_code;
    unsigned short  mouse_bx;
    unsigned short  mouse_cx;
    unsigned short  mouse_dx;
    signed short    mouse_si;
    signed short    mouse_di;
} cbd = { 0 };
```

```
/* Set up data buffer for mouse cursor bitmap */
unsigned short cursor[] = {
    /* 16 words of screen mask */
    0x3fff,  /*0011111111111111*/
    0x1fff,  /*0001111111111111*/
    0x0fff,  /*0000111111111111*/
    0x07ff,  /*0000011111111111*/
    0x03ff,  /*0000001111111111*/
    0x01ff,  /*0000000111111111*/
    0x00ff,  /*0000000011111111*/
    0x007f,  /*0000000001111111*/
    0x01ff,  /*0000000111111111*/
    0x10ff,  /*0001000011111111*/
    0xb0ff,  /*1011000011111111*/
    0xf87f,  /*1111100001111111*/
    0xf87f,  /*1111100001111111*/
    0xfc3f,  /*1111110000111111*/
    0xfc3f,  /*1111110000111111*/
    0xfe1f,  /*1111111000011111*/

    /* 16 words of cursor mask */
    0x0000,  /*0000000000000000*/
    0x4000,  /*0100000000000000*/
    0x6000,  /*0110000000000000*/
    0x7000,  /*0111000000000000*/
    0x7800,  /*0111100000000000*/
    0x7c00,  /*0111110000000000*/
    0x7e00,  /*0111111000000000*/
    0x7f00,  /*0111111100000000*/
    0x7c00,  /*0111110000000000*/
    0x4600,  /*0100011000000000*/
    0x0600,  /*0000011000000000*/
    0x0300,  /*0000001100000000*/
    0x0300,  /*0000001100000000*/
    0x0180,  /*0000000110000000*/
    0x0180,  /*0000000110000000*/
    0x00c0,  /*0000000011000000*/
};

int lock_ region( void *address, unsigned length )
{
    union REGS      regs;
    unsigned        linear;

    /* Thanks to DOS/4GW's zero-based flat memory
       model, converting a pointer of any type to
       a linear address is trivial.
    */
    linear = (unsigned)address;

    /* DPMI Lock Linear Region */
    regs.w.ax = 0x600;
    /* Linear address in BX:CX */
    regs.w.bx = (unsigned short)(linear >> 16);
    regs.w.cx = (unsigned short)(linear & 0xFFFF);
    /* Length in SI:DI */
    regs.w.si = (unsigned short)(length >> 16);
    regs.w.di = (unsigned short)(length & 0xFFFF);
    int386( 0x31, &regs, &regs );
    /* Return 0 if lock failed */
    return( !regs.w.cflag );
}
```

```
#pragma off( check_ stack )
void _ loadds far click_ handler( int max, int mbx,
                                  int mcx, int mdx,
                                  int msi, int mdi )
{
#pragma aux click_ handler _ _ parm [EAX] [EBX] [ECX] \
                                    [EDX] [ESI] [EDI]
    cbd.mouse_ event = 1;

    cbd.mouse_ code = (unsigned short)max;
    cbd.mouse_ bx   = (unsigned short)mbx;
    cbd.mouse_ cx   = (unsigned short)mcx;
    cbd.mouse_ dx   = (unsigned short)mdx;
    cbd.mouse_ si   = (signed short)msi;
    cbd.mouse_ di   = (signed short)mdi;
    if( cbd.mouse_ code & 8 )
        cbd.right_ button = 1;
}

/* Dummy function so we can calculate size of
  code to lock (cbc_ end - click_ handler).
*/
void cbc_ end( void )
{
}
#pragma on( check_ stack )

void main (void)
{
    struct SREGS        sregs;
    union REGS          inregs, outregs;
    int                 installed = 0;
    unsigned char       orig_ mode = 0;
    unsigned short far  *ptr;
    void (far *function_ ptr)();

    segread( &sregs );

    /* get original video mode */

    inregs.w.ax = 0x0f00;
    int386( 0x10, &inregs, &outregs );
    orig_ mode = outregs.h.al;

    /* goto graphics mode */

    inregs.h.ah = 0x00;
    inregs.h.al = 0x6;
    int386( 0x10, &inregs, &outregs );

    printf( "Previous Mode = %u\n", orig_ mode );
    printf( "Current Mode = %u\n", inregs.h.al );

    /* check for mouse driver */

    inregs.w.ax = 0;
    int386( 0x33, &inregs, &outregs );
    if( installed = (outregs.w.ax == 0xffff) )
        printf( "Mouse installed...\n" );
    else
        printf( "Mouse NOT installed...\n" );
```

```
    if( installed ) {
        /* lock callback code and data (essential under VMM!)
           note that click_handler, although it does a far
           return and is installed using a full 48-bit pointer,
           is really linked into the flat model code segment
           -- so we can use a regular (near) pointer in the
           lock_region() call.
        */
        if( (! lock_region( &cbd, sizeof( cbd ) )) ||
            (! lock_region( (void near *)click_handler,
               (char *)cbc_end - (char near *)click_handler )) )
        {
            printf( "locks failed\n" );
        } else {
            /* show mouse cursor */

            inregs.w.ax = 0x1;
            int386( 0x33, &inregs, &outregs );

            /* set mouse cursor form */

            inregs.w.ax  = 0x9;
            inregs.w.bx  = 0x0;
            inregs.w.cx  = 0x0;
            ptr          = cursor;
            inregs.x.edx = FP_OFF( ptr );
            sregs.es     = FP_SEG( ptr );
            int386x( 0x33, &inregs, &outregs, &sregs );

            /* install click watcher */

            inregs.w.ax  = 0xC;
            inregs.w.cx  = 0x0002 + 0x0008;
            function_ptr = ( void (far *)( void ) )click_handler;
            inregs.x.edx = FP_OFF( function_ptr );
            sregs.es     = FP_SEG( function_ptr );
            int386x( 0x33, &inregs, &outregs, &sregs );

            while( !cbd.right_button ) {
                if( cbd.mouse_event ) {
                    printf( "Ev %04hxh  BX %hu  CX %hu  DX %hu  "
                            "SI %hd  DI %hd\n",
                            cbd.mouse_code, cbd.mouse_bx,
                            cbd.mouse_cx, cbd.mouse_dx,
                            cbd.mouse_si, cbd.mouse_di );
                    cbd.mouse_event = 0;
                }
            }
        }
    }

    /* check installation again (to clear watcher) */

    inregs.w.ax = 0;
    int386( 0x33, &inregs, &outregs );
    if( outregs.w.ax == 0xffff )
        printf( "DONE : Mouse still installed...\n" );
    else
        printf( "DONE : Mouse NOT installed...\n" );

    printf( "Press Enter key to return to original mode\n" );
    getc( stdin );
    inregs.h.ah = 0x00;
    inregs.h.al = orig_mode;
    int386( 0x10, &inregs, &outregs );
}
```

5.7 How Do I Simulate a Real-Mode Interrupt with DOS/4GW?

Some interrupts are not supported in protected mode with DOS/4GW but they can still be called using the DPMI function, Simulate Real-Mode Interrupt (0x0300). Information that needs to be passed down to the real-mode interrupt is transferred using an information data structure that is allocated in the protected-mode application. The address to this protected-mode structure is passed into DPMI function 0x0300. DOS/4GW will then use this information to set up the real-mode registers, switch to real mode and then execute the interrupt in real mode.

If your protected-mode application needs to pass data down into the real-mode interrupt, an intermediate real-mode buffer must be used. This buffer can be created using DPMI function 0x0100 to allocate real-mode memory. You can then transfer data from the protected-mode memory to the real-mode memory using a far pointer as illustrated in the "SIMULATE.C" example.

The following example illustrates how to allocate some real-mode memory, transfer a string of characters from protected mode into the real-mode buffer, then set up and call the Interrupt 0x0021 function to create a directory. The string of characters are used to provide the directory name. This example can be adapted to handle most real-mode interrupt calls that aren't supported in protected mode.

```
/*
    SIMULATE.C - Shows how to issue a real-mode interrupt
    from protected mode using DPMI call 300h.  Any buffers
    to be passed to DOS must be allocated in DOS memory
    This can be done with DPMI call 100h.  This program
    will call DOS int 21, function 39h, "Create
    Directory".

    Compile & Link: wcl386 -l=dos4g simulate
*/
#include <i86.h>
#include <dos.h>
#include <stdio.h>
#include <string.h>

static struct rminfo {
    long EDI;
    long ESI;
    long EBP;
    long reserved_by_system;
    long EBX;
    long EDX;
    long ECX;
    long EAX;
    short flags;
    short ES,DS,FS,GS,IP,CS,SP,SS;
} RMI;
void main()
{
    union REGS regs;
    struct SREGS sregs;
    int interrupt_no=0x31;
    short selector;
    short segment;
    char far *str;

    /* DPMI call 100h allocates DOS memory */
    memset(&sregs,0,sizeof(sregs));
    regs.w.ax=0x0100;
    regs.w.bx=0x0001;
    int386x( interrupt_no, &regs, &regs, &sregs);
    segment=regs.w.ax;
    selector=regs.w.dx;
```

```
    /* Move string to DOS real-mode memory */
    str=MK_FP(selector,0);
    _fstrcpy( str, "myjunk" );

    /* Set up real-mode call structure */
    memset(&RMI,0,sizeof(RMI));
    RMI.EAX=0x00003900; /* call service 39h ah=0x39  */
    RMI.DS=segment;     /* put DOS seg:off into DS:DX*/
    RMI.EDX=0;          /* DOS ignores EDX high word */
    /* Use DPMI call 300h to issue the DOS interrupt */
    regs.w.ax = 0x0300;
    regs.h.bl = 0x21;
    regs.h.bh = 0;
    regs.w.cx = 0;
    sregs.es = FP_SEG(&RMI);
    regs.x.edi = FP_OFF(&RMI);
    int386x( interrupt_no, &regs, &regs, &sregs );
}
```

5.8 How do you install a bi-modal interrupt handler using DOS/4GW?

Due to the nature of the protected-mode/real-mode interface, it is often difficult to handle high speed communications with hardware interrupt handlers. For example, if you install your communications interrupt handler in protected mode, you may find that some data is lost when transmitting data from a remote machine at the rate of 9600 baud. This occurs because the data arrived at the communication port while the machine was in the process of transferring the previous interrupt up to protected mode. Data will also be lost if you install the interrupt handler in real mode since your program, running in protected mode, will have to switch down into real mode to handle the interrupt. The reason for this is that the data arrived at the communication port while the DOS extender was switching between real mode and protected mode, and the machine was not available to process the interrupt.

To avoid the delay caused by switching between real-mode and protected mode to handle hardware interrupts, install interrupt handlers in both real-mode and protected-mode. During the execution of a protected-mode program, the system often switches down into real-mode for DOS system calls. If a communications interrupt occurs while the machine is in real-mode, then the real-mode interrupt handler will be used. If the interrupt occurs when the machine is executing in protected-mode, then the protected-mode interrupt handler will be used. This enables the machine to process the hardware interrupts faster and avoid the loss of data caused by context switching.

Installing the interrupt handlers in both protected-mode and real-mode is called bi-modal interrupt handling. The following program is an example of how to install both handlers for Interrupt 0x0C (also known as COM1 or IRQ4). The program writes either a ’P’ to absolute address 0xB8002 or an ’R’ to absolute address 0xB8000. These locations are the first two character positions in screen memory for a color display. As the program runs, you can determine which interrupt is handling the COM1 port by the letter that is displayed. A mouse attached to COM1 makes a suitable demo. Type on the keyboard as you move the mouse around. The ESC key can be used to terminate the program. Transmitted data from a remote machine at 9600 baud can also be used to test the COM1 handling.

```
/*
    BIMODAL.C - The following program demonstrates how
    to set up a bi-modal interrupt handler for DOS/4GW

    Compile and link: wcl386 -l=dos4g bimodal bimo.obj
*/

#include <stdio.h>
#include <conio.h>
#include <dos.h>

#define D32RealSeg(P)   ((((DWORD) (P)) >> 4) & 0xFFFF)
#define D32RealOff(P)   (((DWORD) (P)) & 0xF)

typedef unsigned int WORD;
typedef unsigned long DWORD;

extern void com1_ init (void);
extern void _ _ interrupt pmhandler (void);
extern void _ _ interrupt _ _ far rmhandler (void);
void *D32DosMemAlloc (DWORD size)
{
    union REGS r;

    r.x.eax = 0x0100;           /* DPMI allocate DOS memory */
    r.x.ebx = (size + 15) >> 4; /* Number of paragraphs requested */
    int386 (0x31, &r, &r);

    if( r.x.cflag )  /* Failed */
        return ((DWORD) 0);
    return (void *) ((r.x.eax & 0xFFFF) << 4);
}
void main (void)
{
    union REGS      r;
    struct SREGS    sr;
    void           *lowp;
    void far       *fh;
    WORD            orig_ pm_ sel;
    DWORD           orig_ pm_ off;
    WORD            orig_ rm_ seg;
    WORD            orig_ rm_ off;
    int             c;

/*  Save the starting protected-mode handler address */
    r.x.eax = 0x350C;   /* DOS get vector (INT 0Ch) */
    sr.ds = sr.es = 0;
    int386x (0x21, &r, &r, &sr);
    orig_ pm_ sel = (WORD) sr.es;
    orig_ pm_ off = r.x.ebx;
/*
    Save the starting real-mode handler address using DPMI
    (INT 31h).
*/
    r.x.eax = 0x0200;   /* DPMI get real mode vector */
    r.h.bl = 0x0C;
    int386 (0x31, &r, &r);
    orig_ rm_ seg = (WORD) r.x.ecx;
    orig_ rm_ off = (WORD) r.x.edx;
/*
    Allocate 128 bytes of DOS memory for the real-mode
    handler, which must of course be less than 128 bytes
    long.  Then copy the real-mode handler into that
    segment.
*/
    if(! ( lowp = D32DosMemAlloc(128) ) ) {
        printf ("Couldn't get low memory!\n");
        exit (1);
    }
    memcpy (lowp, (void *) rmhandler, 128);
```

```
/*
    Install the new protected-mode vector.  Because INT 0Ch
    is in the auto-passup range, its normal "passdown"
    behavior will change as soon as we install a
    protected-mode handler.  After this next call, when a
    real mode INT 0Ch is generated, it will be resignalled
    in protected mode and handled by pmhandler.
*/
    r.x.eax = 0x250C;   /* DOS set vector (INT 0Ch) */
    fh = (void far *) pmhandler;
    r.x.edx = FP_OFF (fh);
    /* DS:EDX == &handler */
    sr.ds = FP_SEG (fh);
    sr.es = 0;
    int386x (0x21, &r, &r, &sr);
/*
    Install the new real-mode vector.  We do this after
    installing the protected-mode vector in order to
    override the "passup" behavior.  After the next call,
    interrupts will be directed to the appropriate handler,
    regardless of which mode we are in when they are
    generated.
*/
    r.x.eax = 0x0201;
    r.h.bl = 0x0C;
    /* CX:DX == real mode &handler */
    r.x.ecx = D32RealSeg(lowp);
    r.x.edx = D32RealOff(lowp);
    int386 (0x31, &r, &r);
/*
    Initialize COM1.
*/
    com1_init ();

    puts( "Move mouse, transmit data; ESC to quit\n" );

    while( 1 ) {
        if( kbhit() ) {
            if( ( (c = getch ()) & 0xff ) == 27 )
                    break;
            putch (c);
        }
    delay( 1 );
    }
/*
    Clean up.
*/
    r.x.eax = 0x250C;   /* DOS set vector (INT 0Ch) */
    r.x.edx = orig_pm_off;
    sr.ds = orig_pm_sel;    /* DS:EDX == &handler */
    sr.es = 0;
    int386x (0x21, &r, &r, &sr);

    r.x.eax = 0x0201;   /* DPMI set real mode vector */
    r.h.bl = 0x0C;
    /* CX:DX == real mode &handler */
    r.x.ecx = (DWORD) orig_rm_seg;
    r.x.edx = (DWORD) orig_rm_off;
    int386 (0x31, &r, &r);
}
```

You will also need to create the following assembler code module. The first part provides the interrupt handling routine for the real-mode interrupt handler. The second provides the protected-mode version of the interrupt handler.

```
;**
;** bimo.asm:
;** Assembler code for real-mode and protected-mode
;** INT 0xC interrupt handlers to support the INT 0xC
;** interrupt in both modes
;**
.386
;**
;** The real-mode interrupt handler is in a 16-bit code
;** segment so that the assembler will generate the right
;** code.  We will copy this code down to a 16-bit segment
;** in low memory rather than executing it in place.
;**

_TEXT16 SEGMENT BYTE PUBLIC USE16 'CODE'
    ASSUME  cs:_TEXT16

    PUBLIC  rmhandler_
rmhandler_ :
    push    es
    push    bx
    mov     bx,0B800h
    mov     es,bx                   ; ES = 0xB800
    sub     bx,bx                   ; BX = 0
    mov     WORD PTR es:[bx],0720h  ; Clear 2 char cells
    mov     WORD PTR es:[bx+2],0720h
    mov     BYTE PTR es:[bx],'R'    ; Write R to memory
    pop     bx
    pop     es
    push    ax
    push    dx
    mov     dx,03FAh
    in      al,dx                   ; Read ports so
    mov     dx,03F8h                ; interrupts can
    in      al,dx                   ; continue to be
    mov     dx,020h                 ; generated
    mov     al,dl
    out     dx,al                   ; Send EOI
    pop     dx
    pop     ax
    iret
_TEXT16 ENDS
;**
;** The protected-mode interrupt handler is in a 32-bit code
;** segment.  Even so, we have to be sure to force an IRETD
;** at the end of the handler, because MASM doesn't generate
;** one.  This handler will be called on a 32-bit stack by
;** DOS/4GW.
;**
;** _DATA is the flat model data segment, which we load into
;** ES so we can write to absolute address 0xB8000.  (In the
;** flat model, DS is based at 0.)
;**
_DATA   SEGMENT BYTE PUBLIC USE32 'DATA'
_DATA   ENDS
```

```
DGROUP GROUP _DATA

_TEXT   SEGMENT BYTE PUBLIC USE32 'CODE'
    ASSUME  cs:_TEXT

    PUBLIC  com1_init_
com1_init_ :
    mov     ax,0F3h                 ; 9600,n,8,1
    mov     dx,0                    ; com1
    int     14h                     ; Initialize COM1
    mov     bx,03F8h                ; COM1 port space
    lea     dx,[bx+5]               ; line status reg
    in      al,dx
    lea     dx,[bx+4]               ; modem control reg
    in      al,dx
    or      al,8                    ; enable OUT2 int
    out     dx,al
    lea     dx,[bx+2]               ; int id register
    in      al,dx
    mov     dx,bx                   ; data receive reg
    in      al,dx
    in      al,21h                  ; interrupt mask reg
    and     al,0EFh                 ; force IRQ4 unmask
    out     21h,al
    lea     dx,[bx+1]               ; int enable reg
    mov     al,1
    out     dx,al                   ; enable received int
    ret
    PUBLIC  pmhandler_
pmhandler_ :
    push    es
    push    bx
    mov     bx,DGROUP
    mov     es,bx
    mov     ebx,0B8000h             ; ES:EBX=flat:0B8000h
    mov     DWORD PTR es:[ebx],07200720h  ; Clear cells
    mov     BYTE PTR es:[ebx+2],'P' ; Write P to memory
    pop     bx
    pop     es
    push    ax
    push    dx
    mov     dx,03FAh
    in      al,dx                   ; Read ports so
    mov     dx,03F8h                ; interrupts can
    in      al,dx                   ; continue to be
    mov     dx,020h                 ; generated
    mov     al,dl
    out     dx,al                   ; Send EOI
    pop     dx
    pop     ax
    iretd
_TEXT   ENDS
    END
```

The DOS/4GW DOS Extender

6 The Tenberry Software DOS/4GW DOS Extender

The chapters in this section describe the 32-bit Tenberry Software *DOS/4GW* DOS Extender which is provided with the Open Watcom C/C++ package. *DOS/4GW* is a subset of Tenberry Software's DOS/4G product. *DOS/4GW* is customized for use with the Open Watcom C/C++ package. Key differences are:

- *DOS/4GW* will only execute programs built with a Open Watcom 32-bit compiler such as Open Watcom C/C++ and linked with its run-time libraries.
- The *DOS/4GW* virtual memory manager (VMM), included in the package, is restricted to 32MB of memory.
- *DOS/4GW* does not provide extra functionality such as TSR capability and VMM performance tuning enhancements.

If your application has requirements beyond those provided by *DOS/4GW*, you may wish to acquire DOS/4GW Professional or DOS/4G from:

```
Tenberry Software, Inc.
PO Box 20050
Fountain Hills, Arizona
U.S.A  85269-0050

WWW:    http://www.tenberry.com/dos4g/
Email:  info@tenberry.com
Phone:  1.480.767.8868
Fax:    1.480.767.8709
```

Programs developed to use the restricted version of *DOS/4GW* which is included in the Open Watcom C/C++ package can be distributed on a royalty-free basis, subject to the licensing terms of the product.

7 Linear Executables

To build a linear executable, compile and link it as described in the chapter entitled "Creating 32-bit DOS/4GW Executables". The resulting file will not run independently: you can run it under the Open Watcom Debugger, Tenberry Software Instant-D debugger, or with the standalone "DOS4GW.EXE".

7.1 The Linear Executable Format

DOS/4GW works with files that use the Linear Executable (LE) file format. The format represents a protected-mode program in the context of a 32-bit 386 runtime environment with linear to physical address translation hardware enabled. It uses a flat address space.

This file format is similar to the Segmented Executable (NE) format used in OS/2 1.x and MS Windows. Both support Dynamic Linking, Resources, and are geared toward protected-mode programs. Both formats use tables of "counted ASCII" names, and they use similar relocation formats.

Both formats begin with a DOS style stub program that sophisticated loaders skip. This stub program executes when the *DOS/4GW* loader is not present, displaying the message, *This program cannot run in DOS mode.*

When the Open Watcom Linker is used to link a *DOS/4GW* application, it automatically replaces the default stub program with one that calls DOS4GW.

7.1.1 The Stub Program

The stub at the beginning of a linear executable is a real-mode program that you can modify as you like. For example, you can:

- make the stub program do a checksum on the "DOS4GW.EXE" file to make sure it's the correct version.
- copy protect your program.
- specify a search path for the "DOS4GW.EXE" file.
- add command line arguments.

The SRC directory contains source code for a sample stub program. "WSTUB.C" is a simple example, a good base to start from when you construct your own stub. Please note that you will require a 16-bit C compiler to compile a new stub program. Following is the code in "WSTUB.C":

```
#include <stdio.h>
#include <stdlib.h>
#include <process.h>
#include <errno.h>
#include <string.h>

/* Add environment strings to be searched here */
char *paths_ to_ check[] = {
        "DOS4GPATH",
        "PATH"};

char *dos4g_ path()
{
    static char fullpath[80];
    int i;

    for( i = 0;
         i < sizeof( paths_ to_ check ) / sizeof( paths_ to_ check[0] );
         i++ ) {
        _ searchenv( "dos4gw.exe", paths_ to_ check[i], fullpath );
        if( fullpath[0] ) return( &fullpath );
    }
    for( i = 0;
         i < sizeof( paths_ to_ check ) / sizeof( paths_ to_ check[0] );
         i++ ) {
        _ searchenv( "dos4g.exe", paths_ to_ check[i], fullpath );
        if( fullpath[0] ) return( &fullpath );
    }
    return( "dos4gw.exe" );
}

main( int argc, char *argv[] )
{
    char        *av[4];
    auto char   cmdline[128];

    av[0] = dos4g_ path();              /* Locate the DOS/4G loader */
    av[1] = argv[0];                    /* name of executable to run */
    av[2] = getcmd( cmdline );          /* command line */
    av[3] = NULL;                       /* end of list */
#ifdef QUIET
    putenv( "DOS4G=QUIET" );    /* disables DOS/4G Copyright banner */
#endif
    execvp( av[0], av );
    puts( "Stub exec failed:" );
    puts( av[0] );
    puts( strerror( errno ) );
    exit( 1 );                  /* indicate error */
}
```

7.2 Memory Use

This section explains how a *DOS/4GW* application uses the memory on a 386-based PC/AT. The basic memory layout of an AT machine consists of 640KB of DOS memory, 384KB of upper memory, and an undetermined amount of extended memory. DOS memory and upper memory together compose real memory, the memory that can be addressed when the processor is running in real mode.

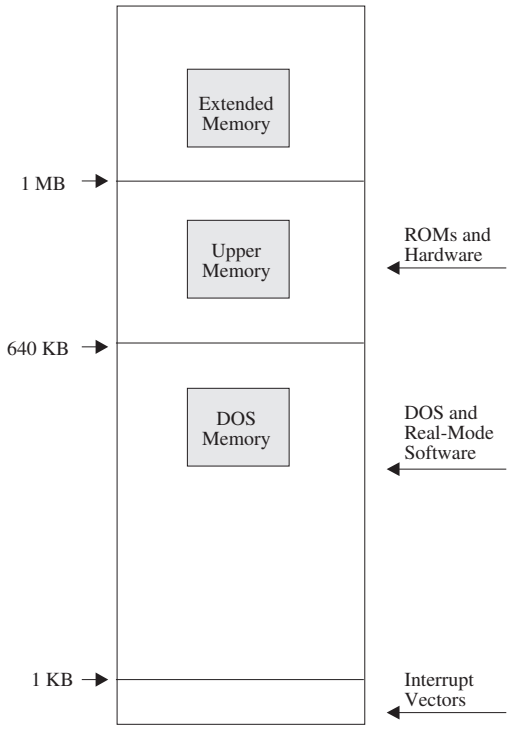


Figure 1. *Basic Memory Layout*

Under *DOS/4GW*, the first megabyte of physical memory — the real memory — is mapped as a shared linear address space. This allows your application to use absolute addresses in real memory, to access video RAM or BIOS ROM, for example. Because the real memory is available to all processes, you are not guaranteed to be able to allocate a particular area in real memory: another process may have allocated it already.

Most code and data is placed in a paged linear address space starting at 4MB. The linear address space starts at 4MB, the first address in the second page table, to avoid conflicts with VCPI system software.

This split mapping — an executable that is linked to start at 4MB in the linear address space, with the first MB in the address space mapped to the first MB of physical memory — is called a *split flat model.*

The illustration below shows the layout of physical memory on the left, and the layout of the linear address space on the right.

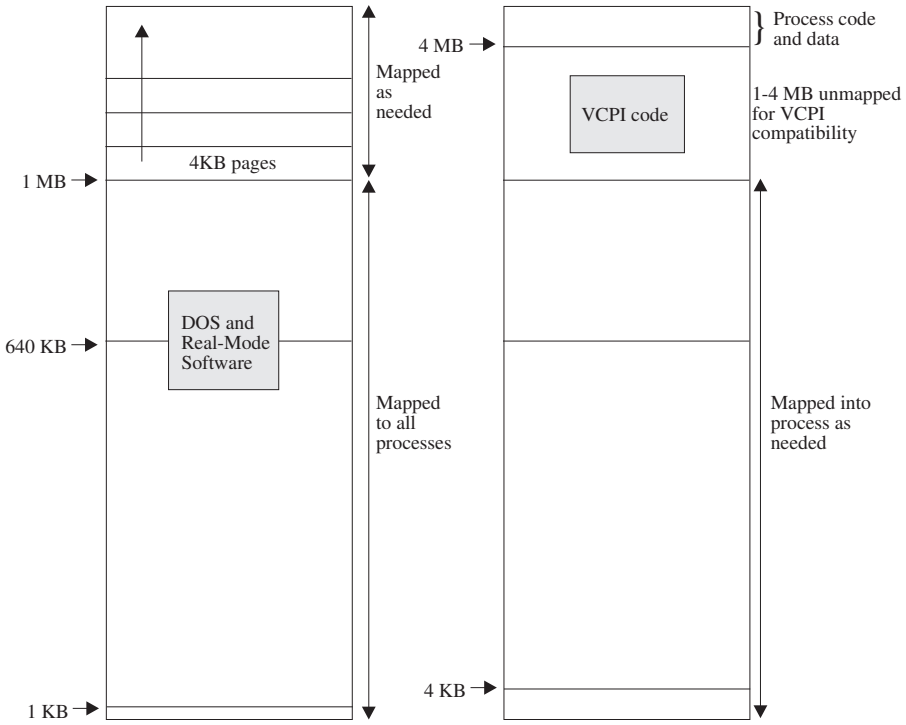


Figure 2. *Physical Memory/Linear Address Space*

The 1KB label in the diagram indicates the top of the real-mode interrupt vectors. 4KB marks the end of the first page.

8 Configuring DOS/4GW

This chapter explains various options that can be specified with the **DOS4G** environment variable including how to suppress the banner that is displayed by *DOS/4GW* at startup. It also explains how to use the **DOS16M** environment variable to select the switch mode setting, if necessary, and to specify the range of extended memory in which *DOS/4GW* will operate. *DOS/4GW* is based on Tenberry Software's DOS/16M 16-bit Protected-Mode support; hence the **DOS16M** environment variable name remains unchanged.

8.1 The DOS4G Environment Variable

A number of options can be selected by setting the **DOS4G** environment variable. The syntax for setting options is:

```
set DOS4G=option1,option2,...
```

Do not insert a space between **DOS4G** and the equal sign. A space to the right of the equal sign is optional.

Options:

QUIET Use this option to suppress the *DOS/4GW* banner.

The banner that is displayed by *DOS/4GW* at startup can be suppressed by issuing the following command:

```
set DOS4G=quiet
```

Note: Use of the quiet switch is only permitted pursuant to the terms and conditions of the WATCOM Software License Agreement and the additional redistribution rights described in the ***Getting Started*** manual. Under these terms, suppression of the copyright by using the quiet switch is not permitted for applications which you distribute to others.

VERBOSE Use this option to maximize the information available for postmortem debugging.

Before running your application, issue the following command:

```
set DOS4G=verbose
```

Reproduce the crash and record the output.

NULLP Use this option to trap references to the first sixteen bytes of physical memory.

Before running your application, issue the following command:

```
set DOS4G=nullp
```

To select a combination of options, list them with commas as separators.

Example:

```
set DOS4G=nullp,verbose
```

8.2 Changing the Switch Mode Setting

In almost all cases, *DOS/4GW* programs can detect the type of machine that is running and automatically choose an appropriate real- to protected-mode switch technique. For the few cases in which this default setting does not work we provide the **DOS16M** DOS environment variable, which overrides the default setting.

Change the switch mode settings by issuing the following command:

```
set DOS16M=value
```

Do not insert a space between **DOS16M** and the equal sign. A space to the right of the equal sign is optional.

The table below lists the machines and the settings you would use with them. Many settings have mnemonics, listed in the column "Alternate Name", that you can use instead of the number. Settings that you must set with the **DOS16M** variable have the notation *req'd* in the first column. Settings you may use are marked *option,* and settings that will automatically be set are marked *auto.*

Status	Machine	Setting	Alternate Name	Comment
auto	386/486 w/ DPMI	0	None	Set automatically if DPMI is active
req'd	NEC 98-series	1	9801	Must be set for NEC 98-series
auto	PS/2	2	None	Set automatically for PS/2
auto	386/486	3	386, 80386	Set automatically for 386 or 486
auto	386	INBOARD	None	386 with Intel Inboard
req'd	Fujitsu FMR-70	5	None	Must be set for Fujitsu FMR-70
auto	386/486 w/ VCPI	11	None	Set automatically if VCPI detected
req'd	Hitachi B32	14	None	Must be set for Hitachi B32
req'd	OKI if800	15	None	Must be set for OKI if800
option	IBM PS/55	16	None	May be needed for some PS/55s

The following procedure shows you how to test the switch mode setting.

1. If you have one of the machines listed below, set the **DOS16M** environment variable to the value shown for that machine and specify a range of extended memory. For example, if your machine is a NEC 98-series, set `DOS16M=1 @2M-4M`. See the section entitled "Fine Control of Memory Usage" on page 49 in this chapter for more information about setting the memory range.

Machine	Setting
NEC 98-series	1
Fujitsu FMR-60,-70	5
Hitachi B32	14
OKI if800	15

Before running *DOS/4GW* applications, check the switch mode setting by following this procedure:

2. Run PMINFO and note the switch setting reported on the last line of the display. (PMINFO, which reports on the protected-mode resources available to your programs, is described in more detail in the chapter entitled "Utilities" on page 87)

   If PMINFO runs, the setting is usable on your machine.

3. If you changed the switch setting, add the new setting to your AUTOEXEC.BAT file.

Note: PMINFO will run successfully on 286 machines. If your *DOS/4GW* application does not run, and PMINFO does, check the CPU type reported on the first line of the display.

You are authorized (and encouraged) to distribute PMINFO to your customers. You may also include a copy of this section in your documentation.

8.3 Fine Control of Memory Usage

In addition to setting the switch mode as described above, the **DOS16M** environment variable enables you to specify which portion of extended memory *DOS/4GW* will use. The variable also allows you to instruct *DOS/4GW* to search for extra memory and use it if it is present.

8.3.1 Specifying a Range of Extended Memory

Normally, you don't need to specify a range of memory with the **DOS16M** variable. You must use the variable, however, in the following cases:

- You are running on a Fujitsu FMR-series, NEC 98-series, OKI if800-series or Hitachi B-series machine.
- You have older programs that use extended memory but don't follow one of the standard disciplines.
- You want to shell out of *DOS/4GW* to use another program that requires extended memory.

If none of these conditions applies to you, you can skip this section.

The general syntax is:

```
set DOS16M= [switch_ mode] [@start_ address [- end_ address]] [:size]
```

In the syntax shown above, `start_ address`,`end_ address` and`size` represent numbers, expressed in decimal or in hexadecimal (hex requires a `0x` prefix). The number may end with a K to indicate an

address or size in kilobytes, or an M to indicate megabytes. If no suffix is given, the address or size is assumed to be in kilobytes. If both a size and a range are specified, the more restrictive interpretation is used.

The most flexible strategy is to specify only a size. However, if you are running with other software that does not follow a convention for indicating its use of extended memory, and these other programs start before *DOS/4GW*, you will need to calculate the range of memory used by the other programs and specify a range for *DOS/4GW* programs to use.

DOS/4GW ignores specifications (or parts of specifications) that conflict with other information about extended memory use. Below are some examples of memory usage control:

set DOS16M= 1 @2m-4m	Mode 1, for NEC 98-series machines, and use extended memory between 2.0 and 4.0MB.
set DOS16M= :1M	Use the last full megabyte of extended memory, or as much as available limited to 1MB.
set DOS16M= @2m	Use any extended memory available above 2MB.
set DOS16M= @ 0 - 5m	Use any available extended memory from 0.0 (really 1.0) to 5.0MB.
set DOS16M= :0	Use no extended memory.

As a default condition *DOS/4GW* applications take all extended memory that is not otherwise in use. Multiple *DOS/4GW* programs that execute simultaneously will share the reserved range of extended memory. Any non-*DOS/4GW* programs started while *DOS/4GW* programs are executing will find that extended memory above the start of the *DOS/4GW* range is unavailable, so they may not be able to run. This is very safe. There will be a conflict only if the other program does not check the BIOS configuration call (Interrupt 15H function 88H, get extended memory size).

To create a private pool of extended memory for your *DOS/4GW* application, use the PRIVATXM program, described in the chapter entitled "Utilities" on page 87.

The default memory allocation strategy is to use extended memory if available, and overflow into DOS (low) memory.

In a VCPI or DPMI environment, the `start_ address` and`end_ address` arguments are not meaningful. *DOS/4GW* memory under these protocols is not allocated according to specific addresses because VCPI and DPMI automatically prevent address conflicts between extended memory programs. You can specify a `size` for memory managed by VCPI or DPMI, but *DOS/4GW* will not necessarily allocate this memory from the highest available extended memory address, as it does for memory managed under other protocols.

8.3.2 Using Extra Memory

Some machines contain extra non-extended, non-conventional memory just below 16MB. When *DOS/4GW* runs on a Compaq 386, it automatically uses this memory because the memory is allocated according to a certain protocol, which *DOS/4GW* follows. Other machines have no protocol for allocating this memory. To use the extra memory that may exist on these machines, set **DOS16M** with the + option.

```
set DOS16M=+
```

Setting the + option causes *DOS/4GW* to search for memory in the range from FA0000 to FFFFFF and determine whether the memory is usable. *DOS/4GW* does this by writing into the extra memory and reading what it has written. In some cases, this memory is mapped for DOS or BIOS usage, or for other system uses. If *DOS/4GW* finds extra memory that is mapped this way, and is not marked read-only, it will write into that memory. This will cause a crash, but won't have any other effect on your system.

8.4 Setting Runtime Options

The **DOS16M** environment variable sets certain runtime options for all *DOS/4GW* programs running on the same system.

To set the environment variable, the syntax is:

```
set DOS16M=[switch_ mode_ setting]^options.
```

Note: Some command line editing TSRs, such as CED, use the caret (^) as a delimiter. If you want to set **DOS16M** using the syntax above while one of these TSRs is resident, modify the TSR to use a different delimiter.

These are the options:

0x01 *check A20 line* -- This option forces *DOS/4GW* to wait until the A20 line is enabled before switching to protected mode. When *DOS/4GW* switches to real mode, this option suspends your program's execution until the A20 line is disabled, unless an XMS manager (such as HIMEM.SYS) is active. If an XMS manager is running, your program's execution is suspended until the A20 line is restored to the state it had when the CPU was last in real mode. Specify this option if you have a machine that runs *DOS/4GW* but is not truly AT-compatible. For more information on the A20 line, see the section entitled "Controlling Address Line 20" on page 52.

0x02 *prevent initialization of VCPI* -- By default, *DOS/4GW* searches for a VCPI server and, if one is present, forces it on. This option is useful if your application does not use EMS explicitly, is not a resident program, and may be used with 386-based EMS simulator software.

0x04 *directly pass down keyboard status calls* -- When this option is set, status requests are passed down immediately and unconditionally. When disabled, pass-downs are limited so the 8042 auxiliary processor does not become overloaded by keyboard polling loops.

0x10 *restore only changed interrupts* -- Normally, when a *DOS/4GW* program terminates, all interrupts are restored to the values they had at the time of program startup. When you use this option, only the interrupts changed by the *DOS/4GW* program are restored.

0x20 *set new memory to 00* -- When *DOS/4GW* allocates a new segment or increases the size of a segment, the memory is zeroed. This can help you find bugs having to do with uninitialized memory. You can also use it to provide a consistent working environment regardless of what programs were run earlier. This option only affects segment allocations or expansions that are made through the *DOS/4GW* kernel (with DOS function 48H or 4AH). This option does not affect memory allocated with a compiler's `malloc` function.

0x40 *set new memory to FF* -- When *DOS/4GW* allocates a new segment or increases the size of a segment, the memory is set to 0xFF bytes. This is helpful in making reproducible cases

of bugs caused by using uninitialized memory. This option only affects segment allocations or expansions that are made through the *DOS/4GW* kernel (with DOS function 48H or 4AH). This option does not affect memory allocated with a compiler's `malloc` function.

0x80 *new selector rotation* -- When *DOS/4GW* allocates a new selector, it usually looks for the first available (unused) selector in numerical order starting with the highest selector used when the program was loaded. When this option is set, the new selector search begins after the last selector that was allocated. This causes new selectors to rotate through the range. Use this option to find references to *stale* selectors, i.e., segments that have been cancelled or freed.

8.5 Controlling Address Line 20

This section explains how *DOS/4GW* uses address line 20 (A20) and describes the related **DOS16M** environment variable settings. It is unlikely that you will need to use these settings.

Because the 8086 and 8088 chips have 20-bit address spaces, their highest addressable memory location is one byte below 1MB. If you specify an address at 1MB or over, which would require a twenty-first bit to set, the address wraps back to zero. Some parts of DOS depend on this wrap, so on the 286 and 386, the twenty-first address bit is disabled. To address extended memory, *DOS/4GW* enables the twenty-first address bit (the A20 line). The A20 line must be enabled for the CPU to run in protected mode, but it may be either enabled or disabled in real mode.

By default, when *DOS/4GW* returns to real mode, it disables the A20 line. Some software depends on the line being enabled. *DOS/4GW* recognizes the most common software in this class, the XMS managers (such as HIMEM.SYS), and enables the A20 line when it returns to real mode if an XMS manager is present. For other software that requires the A20 line to be enabled, use the `A20` option. The `A20` option makes *DOS/4GW* restore the A20 line to the setting it had when *DOS/4GW* switched to protected mode. Set the environment variable as follows:

```
set DOS16M=A20
```

To specify more than one option on the command line, separate the options with spaces.

The **DOS16M** variable also lets you to specify the length of the delay between a *DOS/4GW* instruction to change the status of the A20 line and the next *DOS/4GW* operation. By default, this delay is 1 loop instruction when *DOS/4GW* is running on a 386 machine. In some cases, you may need to specify a longer delay for a machine that will run *DOS/4GW* but is not truly AT-compatible. To change the delay, set **DOS16M** to the desired number of loop instructions, preceded by a comma:

```
set DOS16M=,loops
```

9 VMM

The Virtual Memory Manager (VMM) uses a swap file on disk to augment RAM. With VMM you can use more memory than your machine actually has. When RAM is not sufficient, part of your program is swapped out to the disk file until it is needed again. The combination of the swap file and available RAM is the *virtual memory*.

Your program can use VMM if you set the DOS environment variable, **DOS4GVM**, as follows. To set the **DOS4GVM** environment variable, use the format shown below.

```
set DOS4GVM= [option[#value]] [option[#value]]
```

A "#" is used with options that take values since the DOS command shell will not accept "=".

If you set **DOS4GVM** equal to 1, the default parameters are used for all options.

Example:

```
C>set DOS4GVM=1
```

9.1 VMM Default Parameters

VMM parameters control the options listed below.

MINMEM The minimum amount of RAM managed by VMM. The default is 512KB.

MAXMEM The maximum amount of RAM managed by VMM. The default is 4MB.

SWAPMIN The minimum or initial size of the swap file. If this option is not used, the size of the swap file is based on ***VIRTUALSIZE*** (see below).

SWAPINC The size by which the swap file grows.

SWAPNAME The swap file name. The default name is "DOS4GVM.SWP". By default the file is in the root directory of the current drive. Specify the complete path name if you want to keep the swap file somewhere else.

DELETESWAP Whether the swap file is deleted when your program exits. By default the file is not deleted. Program startup is quicker if the file is not deleted.

VIRTUALSIZE The size of the virtual memory space. The default is 16MB.

9.2 Changing the Defaults

You can change the defaults in two ways.

1. Specify different parameter values as arguments to the **DOS4GVM** environment variable, as shown in the example below.

```
set DOS4GVM=deleteswap maxmem#8192
```

2. Create a configuration file with the filetype extension ".VMC", and use that as an argument to the **DOS4GVM** environment variable, as shown below.

```
set DOS4GVM=@NEW4G.VMC
```

9.2.1 The .VMC File

A ".VMC" file contains VMM parameters and settings as shown in the example below. Comments are permitted. Comments on lines by themselves are preceded by an exclamation point (!). Comments that follow option settings are preceded by white space. Do not insert blank lines: processing stops at the first blank line.

```
!Sample .VMC file
!This file shows the default parameter values.
minmem = 512            At least 512K bytes of RAM is required.
maxmem = 4096           Uses no more than 4MB of RAM
virtualsize = 16384     Swap file plus allocated memory is 16MB
!To delete the swap file automatically when the program exits, add
!deleteswap
!To store the swap file in a directory called SWAPFILE, add
!swapname = c:\swapfile\dos4gvm.swp
```

10 Interrupt 21H Functions

When you call an Interrupt 21H function under *DOS/4GW*, the 32-bit registers in which you pass values are translated into the appropriate 16-bit registers, since DOS works only with 16 bits. However, you can use 32-bit values in your DOS calls. You can allocate blocks of memory larger than 64KB or use an address with a 32-bit offset, and *DOS/4GW* will translate the call appropriately, to use 16-bit registers. When the Interrupt 21H function returns, the value is widened - placed in a 32-bit register, with the high order bits zeroed.

DOS/4GW uses the following rules to manage registers:

- When you pass a parameter to an Interrupt 21H function that expects a 16-bit quantity in a general register (for example, AX), pass a 32-bit quantity in the corresponding extended register (for example, EAX). When a DOS function returns a 16-bit quantity in a general register, expect to receive it (with high-order zero bits) in the corresponding extended register.

- When an Interrupt 21H function expects to receive a 16:16 pointer in a segment:general register pair (for example, ES:BX), supply a 16:32 pointer using the same segment register and the corresponding extended general register (ES:EBX). *DOS/4GW* will copy data and translate pointers so that DOS ultimately receives a 16:16 real-mode pointer in the correct registers.

- When DOS returns a 16:16 real-mode pointer, *DOS/4GW* translates the segment value into an appropriate protected-mode selector and generates a 32-bit offset that results in a 16:32 pointer to the same location in the linear address space.

- Many DOS functions return an error code in AX if the function fails. *DOS/4GW* checks the status of the carry flag, and if it is set, indicating an error, zero-extends the code for EAX. It does not change any other registers.

- If the value is passed or returned in an 8-bit register (AL or AH, for example), *DOS/4GW* puts the value in the appropriate location and leaves the upper half of the 32-bit register untouched.

The table below lists all the Interrupt 21h functions. For each, it shows the registers that are widened or narrowed. Footnotes provide additional information about some of the interrupts that require special handling. Following the table is a section that provides a detailed explanation of interrupt handling under *DOS/4GW*.

Function	Purpose	Managed Registers
00H	Terminate Process	None
01H	Character Input with Echo	None
02H	Character Output	None
03H	Auxiliary Input	None
04H	Auxiliary Output	None
05H	Print Character	None
06H	Direct Console I/O	None
07H	Unfiltered Character Input Without Echo	None
08H	Character Input Without Echo	None
09H	Display String	EDX
0AH	Buffered Keyboard Input	EDX
0BH	Check Keyboard Status	None
0CH	Flush Buffer, Read Keyboard	EDX
0DH	Disk Reset	None
0EH	Select Disk	None
0FH	Open File with FCB	EDX
10H	Close File with FCB	EDX
11H	Find First File	EDX
12H	Find Next File	EDX
13H	Delete File	EDX
14H	Sequential Read	EDX
15H	Sequential Write	EDX
16H	Create File with FCB	EDX
17H	Rename File	EDX
19H	Get Current Disk	None
1AH	Set DTA Address	EDX
1BH	Get Default Drive Data	Returns in EBX, ECX, and EDX
1CH	Get Drive Data	Returns in EBX, ECX, and EDX
21H	Random Read	EDX
22H	Random Write	EDX
23H	Get File Size	EDX
24H	Set Relative Record	EDX
25H	Set Interrupt Vector	EDX
26H	Create New Program Segment Prefix	None
27H	Random Block Read	EDX, returns in ECX
28H	Random Block Write	EDX, returns in ECX
29H	Parse Filename	ESI, EDI, returns in EAX, ESI and EDI (1.)
2AH	Get Date	Returns in ECX
2BH	Set Date	None
2CH	Get Time	None
2DH	Set Time	None
2EH	Set/Reset Verify Flag	None
2FH	Get DTA Address	Returns in EBX
30H	Get MS-DOS Version Number	Returns in ECX
31H	Terminate and Stay Resident	None
33H	Get/Set Control-C Check Flag	None
34H	Return Address of InDOS Flag	Returns in EBX
35H	Get Interrupt Vector	Returns in EBX
36H	Get Disk Free Space	Returns in EAX, EBX, ECX, and EDX

38H	Get/Set Current Country	EDX, returns in EBX
39H	Create Directory	EDX
3AH	Remove Directory	EDX
3BH	Change Current Directory	EDX
3CH	Create File with Handle	EDX, returns in EAX
3DH	Open File with Handle	EDX, returns in EAX
3EH	Close File	None
3FH	Read File or Device	EBX, ECX, EDX, returns in EAX (2.)
40H	Write File or Device	EBX, ECX, EDX, returns in EAX (2.)
41H	Delete File	EDX
42H	Move File Pointer	Returns in EDX, EAX
43H	Get/Set File Attribute	EDX, returns in ECX
44H	IOCTL	(3.)
00H	Get Device Information	Returns in EDX
01H	SetDevice Information	None
02H	Read Control Data from CDD	EDX, returns in EAX
03H	Write Control Data to CDD	EDX, returns in EAX
04H	Read Control Data from BDD	EDX, returns in EAX
05H	Write Control Data to BDD	EDX, returns in EAX
06H	Check Input Status	None
07H	Check Output Status	None
08H	Check if Block Device is Removeable	Returns in EAX
09H	Check if Block Device is Remote	Returns in EDX
0AH	Check if Handle is Remote	Returns in EDX
0BH	Change Sharing Retry Count	None
0CH	Generic I/O Control for Character Devices	EDX
0DH	Generic I/O Control for Block Devices	EDX
0EH	Get Logical Drive Map	None
0FH	Set Logical Drive Map	None
45H	Duplicate File Handle	Returns in EAX
46H	Force Duplicate File Handle	None
47H	Get Current Directory	ESI
48H	Allocate Memory Block	Returns in EAX
49H	Free Memory Block	None
4AH	Resize Memory Block	None
4BH	Load and Execute Program (EXEC)	EBX, EDX (4.)
4CH	Terminate Process with Return Code	None
4DH	Get Return Code of Child Process	None
4EH	Find First File	EDX
4FH	Find Next File	None
52H	Get List of Lists	(not supported)
54H	Get Verify Flag	None
56H	Rename File	EDX, EDI
57H	Get/Set Date/Time of File	Returns in ECX, and EDX
58H	Get/Set Allocation Strategy	Returns in EAX
59H	Get Extended Error Information	Returns in EAX
5AH	Create Temporary File	EDX, returns in EAX and EDX
5BH	Create New File	EDX, returns in EAX
5CH	Lock/Unlock File Region	None
5EH	Network Machine Name/Printer Setup	
00H	Get Machine Name	EDX
02H	Set Printer Setup String	ESI

03H	Get Printer Setup String	EDI, returns in ECX
5FH	Get/Make Assign List Entry	
02H	Get Redirection List Entry	ESI, EDI, returns in ECX
03H	Redirect Device	ESI, EDI
04H	Cancel Device Redirection	ESI
62H	Get Program Segment Prefix Address	Returns in EBX
63H	Get Lead Byte Table (version 2.25 only)	Returns in ESI
65H	Get Extended Country Information	EDI
66H	Get or Set Code Page	None
67H	Set Handle Count	None

This list of functions is excerpted from *The MS-DOS Encyclopedia*, Copyright (c) 1988 by Microsoft Press. All Rights Reserved.

1. For Function 29H, DS:ESI and ES:EDI contain pointer values that are not changed by the call.

2. You can read and write quantities larger than 64KB with Functions 3FH and 40H. *DOS/4GW* breaks your request into chunks smaller than 64KB, and calls the DOS function once for each chunk.

3. You can't transfer more than 64KB using Function 44h, subfunctions 02H, 03H, 04H, or 05H. *DOS/4GW* does not break larger requests into DOS-sized chunks, as it does for Functions 3FH and 40H.

4. When you call Function 4B under *DOS/4GW*, you pass it a data structure that contains 16:32 bit pointers. *DOS/4GW* translates these into 16:16 bit pointers in the structure it passes to DOS.

10.1 Functions 25H and 35H: Interrupt Handling in Protected Mode

By default, interrupts that occur in protected mode are passed down: the entry in the IDT points to code in *DOS/4GW* that switches the CPU to real mode and resignals the interrupt. If you install an interrupt handler using Interrupt 21H, Function 25H, that handler will get control of any interrupts that occur while the processor is in protected mode. If the interrupt for which you installed the handler is in the *autopassup range,* your handler will also get control of interrupts signalled in real mode.

The autopassup range runs from 08H to 2EH inclusive, but excluding 21H. If the interrupt is in the autopassup range, the real-mode vector will be modified when you install the protected-mode handler to point to code in the *DOS/4GW* kernel. This code switches the processor to protected mode and resignals the interrupt-where your protected-mode handler will get control.

10.1.1 32-Bit Gates

The *DOS/4GW* kernel always assigns a 32-bit gate for the interrupt handlers it installs. It does not distinguish between 16-bit and 32-bit handlers for consistency with DPMI.

This 32-bit gate points into the *DOS/4GW* kernel. When *DOS/4GW* handles the interrupt, it switches to its own 16-bit stack, and from there it calls the interrupt handler (yours or the default). This translation is

transparent to the handler, with one exception: since the current stack is not the one on which the interrupt occurred, the handler cannot look up the stack for the address at which the interrupt occurred.

10.1.2 Chaining 16-bit and 32-bit Handlers

If your program hooks an interrupt, write a normal service routine that either handles the interrupt and IRETs or chains to the previous handler. As part of handling the interrupt, your handler can PUSHF/CALL to the previous handler. The handler *must* IRET (or IRETD) or chain.

For each protected-mode interrupt, *DOS/4GW* maintains separate chains of 16-bit and 32-bit handlers. If your 16-bit handler chains, the previous handler is a 16-bit program. If your 32-bit handler chains, the previous handler is a 32-bit program.

If a 16-bit program hooks a given interrupt before any 32-bit programs hook it, the 16-bit chain is executed first. If all the 16-bit handlers unhook later and a new 16-bit program hooks the interrupt while 32-bit handlers are still outstanding, the 32-bit handlers will be executed first.

If the first program to hook an interrupt is 32-bit, the 32-bit chain is executed first.

10.1.3 Getting the Address of the Interrupt Handler

When you signal Interrupt 21H, Function 35, it always returns a non-null address even if no other program of your bitness (i.e., 16-bit or 32-bit) has hooked the interrupt. The address points to a dummy handler that looks to you as though it does an IRET to end the chain. This means that you can't find an unused interrupt by looking for a NULL pointer. Since this technique is most frequently used by programs that are looking for an unclaimed *real-mode* interrupt on which to install a TSR, it shouldn't cause you problems.

11 Interrupt 31H DPMI Functions

When a *DOS/4GW* application runs under a DPMI host, such as Windows 3.1 in enhanced mode, an OS/2 virtual DOS machine, 386Max (with DEBUG=DPMIXCOPY), or QEMM/QDPMI (with EXTCHKOFF), the DPMI host provides the DPMI services, not *DOS/4GW*. The DPMI host also provides virtual memory, if any. Performance (speed and memory use) under different DPMI hosts varies greatly due to the quality of the DPMI implementation.

DPMI services are accessed using Interrupt 31H.

The following describes the services provided by *DOS/4GW* and DOS/4GW Professional in the absence of a DPMI host. *DOS/4GW* supports many of the common DPMI system services. Not all of the services described below are supported by other DPMI hosts.

Some of the information in this chapter was obtained from the the DOS Protected-Mode Interface (DPMI) specification. It is no longer in print; however the DPMI 1.0 specification can be obtained from the Intel ftp site. Here is the URL.

```
ftp://ftp.intel.com/pub/IAL/software_ specs/dpmiv1.zip
```

This ZIP file contains a Postscript version of the DPMI 1.0 specification.

11.1 Using Interrupt 31H Function Calls

Interrupt 31H DPMI function calls can be used only by protected-mode programs.

The general ground rules for Interrupt 31H calls are as follows:

- All Interrupt 31H calls modify the AX register. Unsupported or unsuccessful calls return an error code in AX. Other registers are saved unless they contain specified return values.
- All Interrupt 31H calls modify flags: Unsupported or unsuccessful calls return with the carry flag set. Successful calls clear the carry flag. Only memory management and interrupt flag management calls modify the interrupt flag.
- Memory management calls can enable interrupts.
- All calls are reentrant.

The flag and register information for each call is listed in the following descriptions of supported Interrupt 31H function calls.

11.2 Int31H Function Calls

The Interrupt 31H subfunction calls supported by *DOS/4GW* are listed below by category:

- Local Descriptor Table (LDT) management services
- DOS memory management services
- Interrupt services
- Translation services
- DPMI version
- Memory management services
- Page locking services
- Demand paging performance tuning services
- Physical address mapping
- Virtual interrupt state functions
- Vendor specific extensions
- Coprocessor status

Only the most commonly used Interrupt 31H function calls are supported in this version.

11.2.1 Local Descriptor Table (LDT) Management Services

Function 0000H This function allocates a specified number of descriptors from the LDT and returns the base selector. Pass the following information:

AX = 0000H
CX = number of descriptors to be allocated

If the call succeeds, the carry flag is clear and the base selector is returned in AX. If the call fails, the carry flag is set.

An allocated descriptor is set to the present data type, with a base and limit of zero. The privilege level of an allocated descriptor is set to match the code segment privilege level of the application. To find out the privilege level of a descriptor, use the `lar` instruction.

Allocated descriptors must be filled in by the application. If more than one descriptor is allocated, the returned selector is the first of a contiguous array. Use Function 0003H to get the increment for the next selector in the array.

Function 0001H This function frees the descriptor specified. Pass the following information:

AX = 0001H
BX = the selector to free

Use the selector returned with function 0000h when the descriptor was allocated. To free an array of descriptors, call this function for each descriptor. Use Function 0003H to find out the increment for each descriptor in the array.

If the call succeeds, the carry flag is clear; if it fails, the carry flag is set.

You can use this function to free the descriptors allocated for the program's initial CS, DS, and SS segments, but you should not free other segments that were not allocated with Function 0000H or Function 000DH.

Function 0002H This function converts a real-mode segment to a descriptor that a protected-mode program can address. Pass the following information:

AX = 0002H
BX = real-mode segment address

If the call succeeds, it clears the carry flag and returns the selector mapped to the real-mode segment in AX. If the call fails, the carry flag is set.

If you call this function more than once with the same real-mode segment address, you get the same selector value each time. The descriptor limit is set to 64KB.

The purpose of this function is to give protected-mode programs easy access to commonly used real-mode segments. However, because you cannot modify or free descriptors created by this function, it should be used infrequently. Do not use this function to get descriptors for private data areas.

To examine real-mode addresses using the same selector, first allocate a descriptor, and then use Function 0007H to change the linear base address.

Function 0003H This function returns the increment value for the next selector. Use this function to get the value you add to the base address of an allocated array of descriptors to get the next selector address. Pass the following information:

AX = 0003H

This call always succeeds. The increment value is returned in AX. This value is always a power of two, but no other assumptions can be made.

Function 0006H This function gets the linear base address of a selector. Pass the following information:

AX = 0006H
BX = selector

If the call succeeds, the carry flag is clear and CX:DX contains the 32-bit linear base address of the segment. If the call fails, it sets the carry flag.

If the selector you specify in BX is invalid, the call fails.

Function 0007H This function changes the base address of a specified selector. Only descriptors allocated through Function 0000H should be modified. Pass the following information:

AX = 0007H
BX = selector
CX:DX = the new 32-bit linear base address for the segment

If the call succeeds, the carry flag is clear; if unsuccessful, the carry flag is set.

If the selector you specify in BX is invalid, the call fails.

Function 0008H This function sets the upper limit of a specified segment. Use this function to modify descriptors allocated with Function 0000H only. Pass the following information:

AX = 0008H
BX = selector
CX:DX = 32-bit segment limit

If the call succeeds, the carry flag is clear; if unsuccessful, the carry flag is set.

The call fails if the specified selector is invalid, or if the specified limit cannot be set.

Segment limits greater than 1MB must be page-aligned. This means that limits greater than 1MB must have the low 12 bits set.

To get the limit of a segment, use the 32-bit form of `lsl` for segment limits greater than 64KB.

Function 0009H This function sets the descriptor access rights. Use this function to modify descriptors allocated with Function 0000H only. To examine the access rights of a descriptor, use the `lar` instruction. Pass the following information:

AX = 0009H
BX = selector
CL = Access rights/type byte
CH = 386 extended access rights/type byte

If the call succeeds, the carry flag is clear; if unsuccessful, the carry flag is set. If the selector you specify in BX is invalid, the call fails. The call also fails if the access rights/type byte does not match the format and meet the requirements shown in the figures below.

The access rights/type byte passed in CL has the format shown in the figure below.

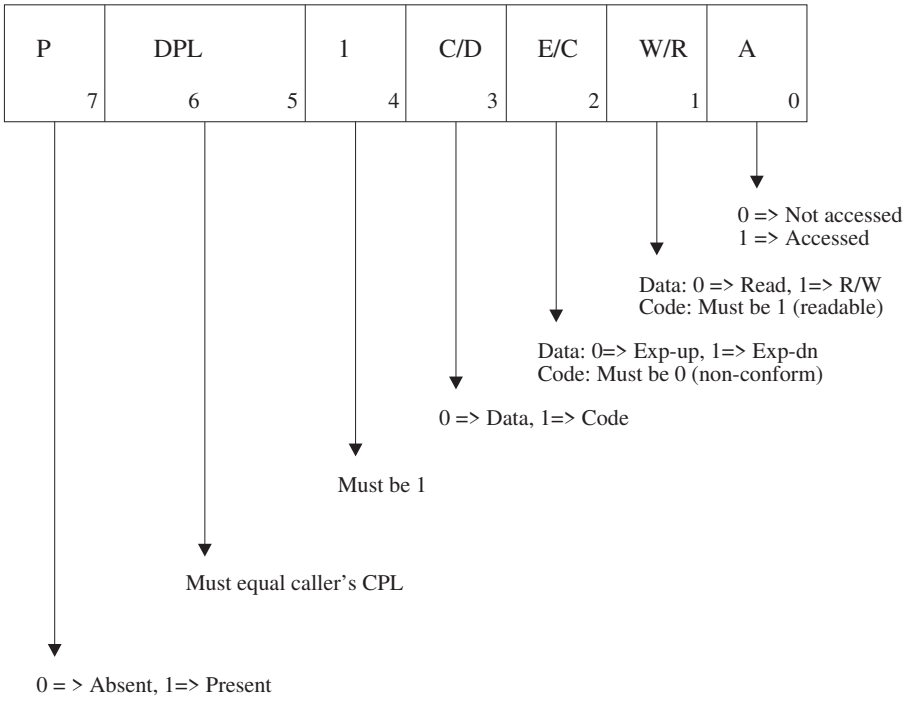


Figure 3. *Access Rights/Type*

The extended access rights/type byte passed in CH has the following format.

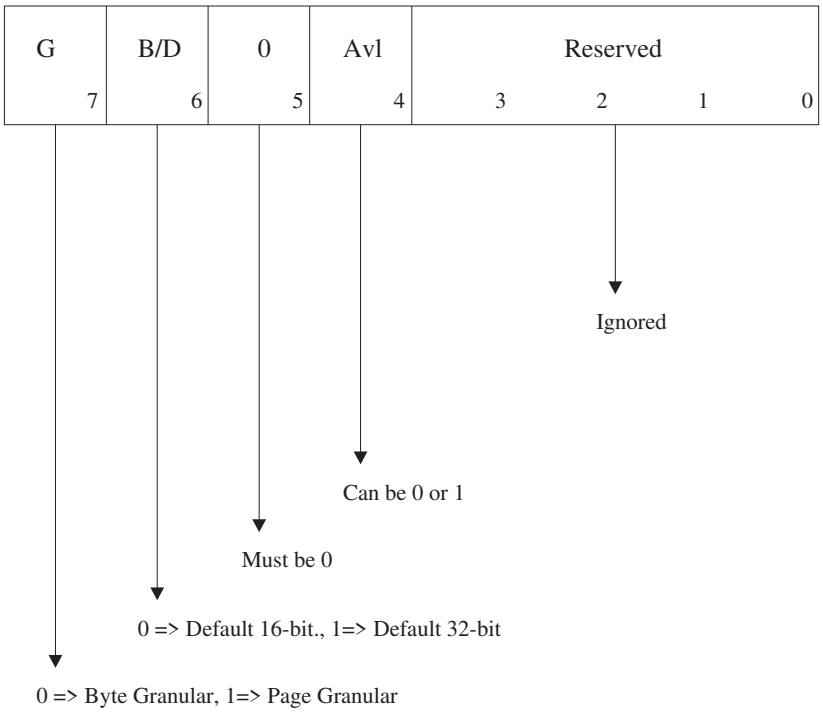


Figure 4. *Extended Access Rights/Type*

Function 000AH This function creates an alias to a code segment. This function creates a data descriptor that has the same base and limit as the specified code segment descriptor. Pass the following information:

AX = 000AH
BX = code segment selector

If the call succeeds, the carry flag is clear and the new data selector is returned in AX. If the call fails, the carry flag is set. The call fails if the selector passed in BX is not a valid code segment.

To deallocate an alias to a code segment, use Function 0001H.

After the alias is created, it does not change if the code segment descriptor changes. For example, if the base or limit of the code segment change later, the alias descriptor stays the same.

Function 000BH This function copies the descriptor table entry for a specified descriptor. The copy is written into an 8-byte buffer. Pass the following information:

AX = 000BH
BX = selector
ES:EDI = a pointer to the 8-byte buffer for the descriptor copy

If the call succeeds, the carry flag is clear and ES:EDI contains a pointer to the buffer that contains a copy of the descriptor. If the call fails, the carry flag is set. The call fails if the selector passed in BX is invalid or unallocated.

Function 000CH This function copies an 8-byte buffer into the LDT for a specified descriptor. The descriptor must first have been allocated with Function 0000H. Pass the following information:

AX = 000CH
BX = selector
ES:EDI = a pointer to the 8-byte buffer containing the descriptor

If the call succeeds, the carry flag is clear; if it fails, the carry flag is set. The call fails if the descriptor passed in BX is invalid.

The type byte, byte 5, has the same format and requirements as the access rights/type byte passed to Function 0009H in CL. The format is shown in the first figure presented with the description of Function 0009H.

The extended type byte, byte 6, has the same format and requirements as the extended access rights/type byte passed to Function 0009H in CH, except that the limit field can have any value, and the low order bits marked *reserved* are used to set the upper 4 bits of the descriptor limit. The format is shown in the second figure presented with the description of Function 0009H.

Function 000DH This function allocates a specific LDT descriptor. Pass the following information:

AX = 000DH
BX = selector

If the call succeeds, the carry flag is clear and the specified descriptor is allocated. If the call fails, the carry flag is set.

The call fails if the specified selector is already in use, or if it is not a valid LDT descriptor. The first 10h (16 decimal) descriptors are reserved for this function, and should not be used by the host. Some of these descriptors may be in use, however, if another client application is already loaded.

To free the descriptor, use Function 0001H.

11.2.2 DOS Memory Management Services

Function 0100H This function allocates memory from the DOS free memory pool. This function returns both the real-mode segment and one or more descriptors that can be used by protected-mode applications. Pass the following information:

AX = 0100H
BX = the number of paragraphs (16-byte blocks) requested

If the call succeeds, the carry flag is clear. AX contains the initial real-mode segment of the allocated block and DX contains the base selector for the allocated block.

If the call fails, the carry flag is set. AX contains the DOS error code. If memory is damaged, code 07H is returned. If there is not enough memory to satisfy the request, code 08H is returned. BX contains the number of paragraphs in the largest available block of DOS memory.

If you request a block larger than 64KB, contiguous descriptors are allocated. Use Function 0003H to find the value of the increment to the next descriptor. The limit of the first descriptor is set to the entire block. Subsequent descriptors have a limit of 64KB, except for the final descriptor, which has a limit of `blocksize MOD 64KB`.

You cannot modify or deallocate descriptors allocated with this function. Function 101H deallocates the descriptors automatically.

Function 0101H This function frees a DOS memory block allocated with function 0100H. Pass the following information:

AX = 0101H
DX = selector of the block to be freed

If the call succeeds, the carry flag is clear.

If the call fails, the carry flag is set and the DOS error code is returned in AX. If the incorrect segment was specified, code 09H is returned. If memory control blocks are damaged, code 07H is returned.

All descriptors allocated for the specified memory block are deallocated automatically and cannot be accessed correctly after the block is freed.

Function 0102H This function resizes a DOS memory block allocated with function 0100H. Pass the following information:

AX = 0102H
BX = the number of paragraphs (16-byte blocks) in the resized block
DX = selector of block to resize

If the call succeeds, the carry flag is clear.

If the call fails, the carry flag is set, the maximum number of paragraphs available is returned in BX, and the DOS error code is returned in AX. If memory code blocks are damaged, code 07H is returned. If there isn't enough memory to increase the size as requested, code 08H is returned. If the incorrect segment is specified, code 09h is returned.

Because of the difficulty of finding additional contiguous memory or descriptors, this function is not often used to increase the size of a memory block. Increasing the size of a memory block might well fail because other DOS allocations have used contiguous space. If the next descriptor in the LDT is not free, allocation also fails when the size of a block grows over the 64KB boundary.

If you shrink the size of a memory block, you may also free some descriptors allocated to the block. The initial selector remains unchanged, however; only the limits of subsequent selectors will change.

11.2.3 Interrupt Services

Function 0200H This function gets the value of the current task's real-mode interrupt vector for the specified interrupt. Pass the following information:

AX = 0200H
BL = interrupt number

This call always succeeds. All 100H (256 decimal) interrupt vectors are supported by the host. When the call returns, the carry flag is clear, and the `segment:offset` of the real-mode interrupt handler is returned in CX:DX.

Because the address returned in CX is a segment, and not a selector, you cannot put it into a protected-mode segment register. If you do, a general protection fault may occur.

Function 0201H This function sets the value of the current task's real-mode interrupt vector for the specified interrupt. Pass the following information:

AX = 0201H
BL = interrupt number
CX:DX = segment:offset of the real-mode interrupt handler

If the call succeeds, the carry flag is clear; if it fails, the carry flag is set.

The address passed in CX:DX should be a real-mode `segment:offset,` such as function 0200H returns. For this reason, the interrupt handler must reside in DOS addressable memory. You can use Function 0100H to allocate DOS memory. This version does not support the real-mode callback address function.

If you are hooking a hardware interrupt, you have to lock all segments involved. These segments include the segment in which the interrupt handler runs, and any segment it may touch at interrupt time.

Function 0202H This function gets the processor exception handler vector. This function returns the CS:EIP of the current protected-mode exception handler for the specified exception number. Pass the following information:

AX = 0202H
BL = exception/fault number (00h - 1Fh)

If the call succeeds, the carry flag is clear and the `selector:offset` of the protected-mode exception handler is returned in CX:EDX. If it fails, the carry flag is set.

The value returned in CX is a valid protected-mode selector, not a real-mode segment.

Function 0203H This function sets the processor exception handler vector. This function allows protected-mode applications to intercept processor exceptions that are not handled by the DPMI environment. Programs may wish to handle exceptions such as "not present segment faults" which would otherwise generate a fatal error. Pass the following information:

AX = 0203H
BL = exception/fault number (00h - 1Fh)
CX:EDX = selector:offset of the exception handler

If the call succeeds, the carry flag is clear. If it fails, the carry flag is set.

The address passed in CX must be a valid protected-mode selector, such as Function 204H returns, and not a real-mode segment. A 32-bit implementation must supply a 32-bit offset in the EDX register. If the handler chains to the next handler, it must use a 32-bit interrupt stack frame to do so.

The handler should return using a far return instruction. The original SS:ESP, CS:EIP and flags on the stack, including the interrupt flag, will be restored.

All fault stack frames have an error code. However the error code is only valid for exceptions 08h, 0Ah, 0Bh, 0Ch, 0Dh, and 0Eh.

The handler must preserve and restore all registers.

The exception handler will be called on a locked stack with interrupts disabled. The original SS, ESP, CS, and EIP will be pushed on the exception handler stack frame.

The handler must either return from the call by executing a far return or jump to the next handler in the chain (which will execute a far return or chain to the next handler).

The procedure can modify any of the values on the stack pertaining to the exception before returning. This can be used, for example, to jump to a procedure by modifying the CS:EIP on the stack. Note that the procedure must not modify the far return address on the stack — it must return to the original caller. The caller will then restore the flags, CS:EIP and SS:ESP from the stack frame.

If the DPMI client does not handle an exception, or jumps to the default exception handler, the host will reflect the exception as an interrupt for exceptions 0, 1, 2, 3, 4, 5 and 7. Exceptions 6 and 8 - 1Fh will be treated as fatal errors and the client will be terminated.

Exception handlers will only be called for exceptions that occur in protected mode.

Function 0204H This function gets the CS:EIP `selector:offset` of the current protected-mode interrupt handler for a specified interrupt number. Pass the following information:

AX = 0204H
BL = interrupt number

This call always succeeds. All 100H (256 decimal) interrupt vectors are supported by the host. When the call returns, the carry flag is clear and CX:EDX contains the protected-mode `selector:offset` of the exception handler.

A 32-bit offset is returned in the EDX register.

Function 0205H This function sets the address of the specified protected-mode interrupt vector. Pass the following information:

AX = 0205H
BL = interrupt number
CX:EDX = selector:offset of the exception handler

If the call succeeds, the carry flag is clear; if it fails, the carry flag is set.

The address passed in CX must be a valid protected-mode selector, such as Function 204H returns, and not a real-mode segment. A 32-bit implementation must supply a 32-bit offset in the EDX register. If the handler chains to the next handler, it must use a 32-bit interrupt stack frame to do so.

11.2.4 Translation Services

These services are provided so that protected-mode programs can call real-mode software that DPMI does not support directly. The protected-mode program must set up a data structure with the appropriate register values. This "real-mode call structure" is shown below.

Offset	Register
00H	EDI
04H	ESI
08H	EBP
0CH	Reserved by system
10H	EBX
14H	EDX
18H	ECX
1CH	EAX
20H	Flags
22H	ES
24H	DS
26H	FS
28H	GS
2AH	IP
2CH	CS
2EH	SP
30H	SS

After the call or interrupt is complete, all real-mode registers and flags except SS, SP, CS, and IP will be copied back to the real-mode call structure so that the caller can examine the real-mode return values.

The values in the segment registers should be real-mode segments, not protected-mode selectors.

The translation services will provide a real-mode stack if the SS:SP fields are zero. However, the stack provided is relatively small. If the real-mode procedure/interrupt routine uses more than 30 words of stack space then you should provide your own real-mode stack.

Function 0300H This function simulates a real-mode interrupt. This function simulates an interrupt in real mode. It will invoke the CS:IP specified by the real-mode interrupt vector and the handler must return by executing an `iret`. Pass the following information:

AX = 0300H
BL = interrupt number
BH = flags Bit 0 = 1 resets the interrupt controller and A20 line. Other flags are reserved and must be 0.
CX = number of words to copy from protected-mode stack to real-mode stack
ES:EDI = the selector:offset of real-mode call structure

If the call fails, the carry flag is set.

If the call succeeds, the carry flag is clear and ES:EDI contains the `selector:offset` of the modified real-mode call structure.

The CS:IP in the real-mode call structure is ignored by this service. The appropriate interrupt handler will be called based on the value passed in BL.

The flags specified in the real-mode call structure will be pushed on the real-mode stack `iret` frame. The interrupt handler will be called with the interrupt and trace flags clear.

It is up to the caller to remove any parameters that were pushed on the protected-mode stack.

The flag to reset the interrupt controller and the A20 line is ignored by DPMI implementations that run in Virtual 8086 mode. It causes DPMI implementations that return to real mode to set the interrupt controller and A20 address line hardware to its normal real-mode state.

Function 0301H (DOS/4GW Professional only) This function calls a real-mode procedure with a FAR return frame. The called procedure must execute a FAR return when it completes. Pass the following information:

AX = 0301H
BH = flags Bit 0 = 1 resets the interrupt controller and A20 line. Other flags reserved and must be 0.
CX = Number of words to copy from protected-mode to real-mode stack
ES:EDI = selector:offset of real-mode call structure

If the call succeeds, the carry flag is clear and ES:EDI contains the `selector:offset` of modified real-mode call structure.

If the call fails, the carry flag is set.

Notes:

1. The CS:IP in the real-mode call structure specifies the address of the real-mode procedure to call.

2. The real-mode procedure must execute a FAR return when it has completed.

3. If the SS:SP fields are zero then a real-mode stack will be provided by the DPMI host. Otherwise, the real-mode SS:SP will be set to the specified values before the procedure is called.

4. When the Int 31h returns, the real-mode call structure will contain the values that were returned by the real-mode procedure.

5. It is up to the caller to remove any parameters that were pushed on the protected-mode stack.

6. The flag to reset the interrupt controller and A20 line is ignored by DPMI implementations that run in Virtual 8086 mode. It causes DPMI implementations that return to real mode to set the interrupt controller and A20 address line hardware to its normal real-mode state.

Function 0302H (DOS/4GW Professional only) This function calls a real-mode procedure with an `iret` frame. The called procedure must execute an `iret` when it completes. Pass the following information:

AX = 0302H
BH = flags Bit 0 = 1 resets the interrupt controller and A20 line. Other flags reserved and must be 0.
CX = Number of words to copy from protected-mode to real-mode stack
ES:EDI = selector:offset of real-mode call structure

If the call succeeds, the carry flag is clear and ES:EDI contains the `selector:offset` of modified real-mode call structure.

If the call fails, the carry flag is set.

Notes:

1. The CS:IP in the real-mode call structure specifies the address of the real-mode procedure to call.

2. The real-mode procedure must execute an `iret` when it has completed.

3. If the SS:SP fields are zero then a real-mode stack will be provided by the DPMI host. Otherwise, the real-mode SS:SP will be set to the specified values before the procedure is called.

4. When the Int 31h returns, the real-mode call structure will contain the values that were returned by the real-mode procedure.

5. The flags specified in the real-mode call structure will be pushed the real-mode stack `iret` frame. The procedure will be called with the interrupt and trace flags clear.

6. It is up to the caller to remove any parameters that were pushed on the protected-mode stack.

7. The flag to reset the interrupt controller and A20 line is ignored by DPMI implementations that run in Virtual 8086 mode. It causes DPMI implementations that return to real mode to set the interrupt controller and A20 address line hardware to its normal real-mode state.

Function 0303H (DOS/4GW Professional only) This function allocates a real-mode callback address. This service is used to obtain a unique real-mode SEG:OFFSET that will transfer control from real mode to a protected-mode procedure.

At times it is necessary to hook a real-mode interrupt or device callback in a protected-mode driver. For example, many mouse drivers call an address whenever the mouse is moved. Software running in protected mode can use a real-mode callback to intercept the mouse driver calls. Pass the following information:

AX = 0303H
DS:ESI = selector:offset of procedure to call
ES:EDI = selector:offset of real-mode call structure

If the call succeeds, the carry flag is clear and CX:DX contains the `segment:offset` of real-mode callback address.

If the call fails, the carry flag is set.

Callback Procedure Parameters

Interrupts disabled
DS:ESI = selector:offset of real-mode SS:SP
ES:EDI = selector:offset of real-mode call structure
SS:ESP = Locked protected-mode API stack
All other registers undefined

Return from Callback Procedure

Execute an IRET to return
ES:EDI = selector:offset of real-mode call structure
to restore (see note)

Notes:

1. Since the real-mode call structure is static, you must be careful when writing code that may be reentered. The simplest method of avoiding reentrancy is to leave interrupts disabled throughout the entire call. However, if the amount of code executed by the callback is large then you will need to copy the real-mode call structure into another buffer. You can then return with ES:EDI pointing to the buffer you copied the data to — it does not have to point to the original real mode call structure.

2. The called procedure is responsible for modifying the real-mode CS:IP before returning. If the real-mode CS:IP is left unchanged then the real-mode callback will be executed immediately and your procedure will be called again. Normally you will want to pop a return address off of the real-mode stack and place it in the real-mode CS:IP. The example code in the next section demonstrates chaining to another interrupt handler and simulating a real-mode `iret.`

3. To return values to the real-mode caller, you must modify the real-mode call structure.

4. Remember that all segment values in the real-mode call structure will contain real-mode segments, not selectors. If you need to examine data pointed to by a real-mode seg:offset pointer, you should not use the segment to selector service to create a new selector. Instead, allocate a descriptor during initialization and change the descriptor's base to 16 times the real-mode segment's value. This is

important since selectors allocated though the segment to selector service can never be freed.

5. DPMI hosts should provide a minimum of 16 callback addresses per task.

The following code is a sample of a real-mode interrupt hook. It hooks the DOS Int 21h and returns an error for the delete file function (AH=41h). Other calls are passed through to DOS. This example is somewhat silly but it demonstrates the techniques used to hook a real mode interrupt. Note that since DOS calls are reflected from protected mode to real mode, the following code will intercept all DOS calls from both real mode and protected mode.

```
;*****************************************************
; This procedure gets the current Int 21h real-mode
; Seg:Offset, allocates a real-mode callback address,
; and sets the real-mode Int 21h vector to the call-
; back address.
;*****************************************************
Initialization_ Code:
;
; Create a code segment alias to save data in
;
        mov     ax, 000Ah
        mov     bx, cs
        int     31h
        jc      ERROR
        mov     ds, ax
        ASSUMES DS,_ TEXT
;
; Get current Int 21h real-mode SEG:OFFSET
;
        mov     ax, 0200h
        mov     bl, 21h
        int     31h
        jc      ERROR
        mov     [Orig_ Real_ Seg], cx
        mov     [Orig_ Real_ Offset], dx
;
; Allocate a real-mode callback
;
        mov     ax, 0303h
        push    ds
        mov     bx, cs
        mov     ds, bx
        mov     si, OFFSET My_ Int_ 21_ Hook
        pop     es
        mov     di, OFFSET My_ Real_ Mode_ Call_ Struc
        int     31h
        jc      ERROR
;
; Hook real-mode int 21h with the callback address
;
        mov     ax, 0201h
        mov     bl, 21h
        int     31h
        jc      ERROR

;*****************************************************
;
; This is the actual Int 21h hook code.  It will return
; an "access denied" error for all calls made in real
; mode to delete a file.  Other calls will be passed
; through to DOS.
;
; ENTRY:
;    DS:SI -> Real-mode SS:SP
;    ES:DI -> Real-mode call structure
;    Interrupts disabled
;
; EXIT:
;    ES:DI -> Real-mode call structure
;
;*****************************************************

My_ Int_ 21_ Hook:
        cmp     es:[di.RealMode_ AH], 41h
        jne     Chain_ To_ DOS
;
; This is a delete file call (AH=41h).  Simulate an
; iret on the real-mode stack, set the real-mode
; carry flag, and set the real-mode AX to 5 to indicate
; an access denied error.
;
```

```
        cld
        lodsw                   ; Get real-mode ret IP
        mov     es:[di.RealMode_ IP], ax
        lodsw                   ; Get real-mode ret CS
        mov     es:[di.RealMode_ CS], ax
        lodsw                   ; Get real-mode flags
        or      ax, 1           ; Set carry flag
        mov     es:[di.RealMode_ Flags], ax
        add     es:[di.RealMode_ SP], 6
        mov     es:[di.RealMode_ AX], 5
        jmp     My_ Hook_ Exit
;
; Chain to original Int 21h vector by replacing the
; real-mode CS:IP with the original Seg:Offset.
;
Chain_ To_ DOS:
        mov     ax, cs:[Orig_ Real_ Seg]
        mov     es:[di.RealMode_ CS], ax
        mov     ax, cs:[Orig_ Real_ Offset]
        mov     es:[di.RealMode_ IP], ax

My_ Hook_ Exit:
        iret
```

Function 0304H (DOS/4GW Professional only) This function frees a real-mode callback address that was allocated through the allocate real-mode callback address service. Pass the following information:

AX = 0304H
CX:DX = Real-mode callback address to free

If the call succeeds, the carry flag is clear; if it fails, the carry flag is set.

Notes:

1. Real-mode callbacks are a limited resource. Your code should free any break point that it is no longer using.

11.2.5 DPMI Version

Function 0400H This function returns the version of DPMI services supported. Note that this is not necessarily the version of any operating system that supports DPMI. It should be used by programs to determine what calls are legal in the current environment. Pass the following information:

AX = 0400H

The information returned is:

AH = Major version
AL = Minor version
BX = Flags Bit 0 = 1 if running under an 80386 DPMI implementation. Bit 1 = 1 if processor is returned to real mode for reflected interrupts (as opposed to Virtual 8086 mode). Bit 2 = 1 if virtual memory is supported. Bit 3 is reserved and undefined. All other bits are zero and reserved for later use.
CL = Processor type

02 = 80286
03 = 80386
04 = 80486
05 = Pentium

DH = Current value of virtual master PIC base interrupt
DL = Current value of virtual slave PIC base interrupt
Carry flag clear (call cannot fail)

11.2.6 Memory Management Services

Function 0500H This function gets information about free memory. Pass the following information:

AX = 0500H
ES:EDI = the selector:offset of a 30H byte buffer.

If the call fails, the carry flag is set.

If the call succeeds, the carry flag is clear and ES:EDI contains the `selector:offset` of a buffer with the structure shown in the figure below.

Offset	Description
00H	Largest available block, in bytes
04H	Maximum unlocked page allocation
08H	Largest block of memory (in pages) that could be allocated and then locked
0CH	Total linear address space size, in pages, including already allocated pages
10H	Total number of free pages and pages currently unlocked and available for paging out
14H	Number of physical pages not in use
18H	Total number of physical pages managed by host
1CH	Free linear address space, in pages
20H	Size of paging/file partition, in pages
24H - 2FH	Reserved

Only the first field of the structure is guaranteed to contain a valid value. Any field that is not returned by *DOS/4GW* is set to -1 (0FFFFFFFFH).

Function 0501H This function allocates and commits linear memory. Pass the following information:

AX = 0501H
BX:CX = size of memory to allocate, in bytes.

If the call succeeds, the carry flag is clear, BX:CX contains the linear address of the allocated memory, and SI:DI contains the memory block handle used to free or resize the block. If the call fails, the carry flag is set.

No selectors are allocated for the memory block. The caller must allocate and initialize selectors needed to access the memory.

If VMM is present, the memory is allocated as unlocked, page granular blocks. Because of the page granularity, memory should be allocated in multiples of 4KB.

Function 0502H This function frees a block of memory allocated through function 0501H. Pass the following information:

AX = 0502H
SI:DI = handle returned with function 0501H when memory was allocated

If the call succeeds, the carry flag is clear; if it fails, the carry flag is set. You must also free any selectors allocated to point to the freed memory block.

Function 0503H This function resizes a block of memory allocated through the 0501H function. If you resize a block of linear memory, it may have a new linear address and a new handle. Pass the following information:

AX = 0503H
BX:CX = new size of memory block, in bytes
SI:DI = handle returned with function 0501H when memory was allocated

If the call succeeds, the carry flag is clear, BX:CX contains the new linear address of the memory block, and SI:DI contains the new handle of the memory block. If the call fails, the carry flag is set.

If either the linear address or the handle has changed, update the selectors that point to the memory block. Use the new handle instead of the old one.

You cannot resize a memory block to zero bytes.

11.2.7 Page Locking Services

These services are only useful under DPMI implementations that support virtual memory. Although memory ranges are specified in bytes, the actual unit of memory that will be locked will be one or more pages. Page locks are maintained as a count. When the count is decremented to zero, the page is unlocked and can be swapped to disk. This means that if a region of memory is locked three times then it must be unlocked three times before the pages will be unlocked.

Function 0600H This function locks a specified linear address range. Pass the following information:

AX = 0600H
BX:CX = starting linear address of memory to lock
SI:DI = size of region to lock (in bytes)

If the call fails, the carry flag is set and none of the memory will be locked.

If the call succeeds, the carry flag is clear. If the specified region overlaps part of a page at the beginning or end of a region, the page(s) will be locked.

Function 0601H This function unlocks a specified linear address range that was previously locked using the "lock linear region" function (0600h). Pass the following information:

AX = 0601H
BX:CX = starting linear address of memory to unlock
SI:DI = size of region to unlock (in bytes)

If the call fails, the carry flag is set and none of the memory will be unlocked. An error will be returned if the memory was not previously locked or if the specified region is invalid.

If the call succeeds, the carry flag is clear. If the specified region overlaps part of a page at the beginning or end of a region, the page(s) will be unlocked. Even if the call succeeds, the memory will remain locked if the lock count is not decremented to zero.

Function 0604H This function gets the page size for Virtual Memory (VM) only. This function returns the size of a single memory page in bytes. Pass the following information:

AX = 0604H

If the call succeeds, the carry flag is clear and BX:CX = Page size in bytes.

If the call fails, the carry flag is set.

11.2.8 Demand Paging Performance Tuning Services

Some applications will discard memory objects or will not access objects for long periods of time. These services can be used to improve the performance of demand paging.

Although these functions are only relevant for DPMI implementations that support virtual memory, other implementations will ignore these functions (it will always return carry clear). Therefore your code can always call these functions regardless of the environment it is running under.

Since both of these functions are simply advisory functions, the operating system may choose to ignore them. In any case, your code should function properly even if the functions fail.

Function 0702H (DOS/4GW Professional only) This function marks a page as a demand paging candidate. This function is used to inform the operating system that a range of pages should be placed at the head of the page out candidate list. This will force these pages to be swapped to disk ahead of other pages even if the memory has been accessed recently. However, all memory contents will be preserved.

This is useful, for example, if a program knows that a given piece of data will not be accessed for a long period of time. That data is ideal for swapping to disk since the physical memory it now occupies can be used for other purposes. Pass the following information:

AX = 0702H
BX:CX = Starting linear address of pages to mark
SI:DI = Number of bytes to mark as paging candidates

If the call succeeds, the carry flag is clear; if it fails, the carry flag is set.

Notes:

1. This function does not force the pages to be swapped to disk immediately.

2. Partial pages will not be discarded.

Function 0703H (DOS/4GW Professional only) This function discards page contents. This function discards the entire contents of a given linear memory range. It is used after a memory object that occupied a given piece of memory has been discarded.

The contents of the region will be undefined the next time the memory is accessed. All values previously stored in this memory will be lost. Pass the following information:

AX = 0703H
BX:CX = Starting linear address of pages to discard
SI:DI = Number of bytes to discard

If the call succeeds, the carry flag is clear; if it fails, the carry flag is set.

Notes:

1. Partial pages will not be discarded.

11.2.9 Physical Address Mapping

Memory mapped devices such as network adapters and displays sometimes have memory mapped at physical addresses that lie outside of the normal 1Mb of memory that is addressable in real mode. Under many implementations of DPMI, all addresses are linear addresses since they use the paging mechanism of the 80386. This service can be used by device drivers to convert a physical address into a linear address. The linear address can then be used to access the device memory.

Function 0800H This function is used for Physical Address Mapping.

Some implementations of DPMI may not support this call because it could be used to circumvent system protection. This call should only be used by programs that absolutely require direct access to a memory mapped device.

Pass the following information:

AX = 0800H
BX:CX = Physical address of memory
SI:DI = Size of region to map in bytes

If the call succeeds, the carry flag is clear and BX:CX = Linear Address that can be used to access the physical memory.

If the call fails, the carry flag is set.

Notes:

1. Under DPMI implementations that do not use the 80386 paging mechanism, the call will always succeed and the address returned will be equal to the physical address parameter passed into this function.

2. It is up to the caller to build an appropriate selector to access the memory.

3. Do not use this service to access memory that is mapped in the first megabyte of address space (the real-mode addressable region).

Function 0801H This function is used to free Physical Address Mapping. Pass the following information:

AX = 0801H
BX:CX = Linear address returned by Function 0800H.

If the call succeeds, the carry flag is clear; if it fails, the carry flag is set.

Notes:

1. The client should call this function when it is finished using a device previously mapped to linear addresses with the Physical Address Mapping function (Function 0800H).

11.2.10 Virtual Interrupt State Functions

Under many implementations of DPMI, the interrupt flag in protected mode will always be set (interrupts enabled). This is because the program is running under a protected operating system that cannot allow programs to disable physical hardware interrupts. However, the operating system will maintain a "virtual" interrupt state for protected-mode programs. When the program executes a CLI instruction, the program's virtual interrupt state will be disabled, and the program will not receive any hardware interrupts until it executes an STI to reenable interrupts (or calls service 0901h).

When a protected-mode program executes a PUSHF instruction, the real processor flags will be pushed onto the stack. Thus, examining the flags pushed on the stack is not sufficient to determine the state of the program's virtual interrupt flag. These services enable programs to get and modify the state of their virtual interrupt flag.

The following sample code enters an interrupt critical section and then restores the virtual interrupt state to it's previous state.

```
;
; Disable interrupts and get previous interrupt state
;
        mov     ax, 0900h
        int     31h
;
; At this point AX = 0900h or 0901h
;
        .
        .
        .
;
; Restore previous state (assumes AX unchanged)
;
        int     31h
```

Function 0900H This function gets and disables Virtual Interrupt State. This function will disable the virtual interrupt flag and return the previous state of the virtual interrupt flag. Pass the following information:

AX = 0900H

After the call, the carry flag is clear (this function always succeeds) and virtual interrupts are disabled.

AL = 0 if virtual interrupts were previously disabled.
AL = 1 if virtual interrupts were previously enabled.

Notes:

1. AH will not be changed by this procedure. Therefore, to restore the previous state, simply execute an Int 31h.

Function 0901H This function gets and enables the Virtual Interrupt State. This function will enable the virtual interrupt flag and return the previous state of the virtual interrupt flag. Pass the following information:

AX = 0901H

After the call, the carry flag is clear (this function always succeeds) and virtual interrupts are enabled.

AL = 0 if virtual interrupts were previously disabled.
AL = 1 if virtual interrupts were previously enabled.

Notes:

1. AH will not be changed by this procedure. Therefore, to restore the previous state, simply execute an Int 31h.

Function 0902H This function gets the Virtual Interrupt State. This function will return the current state of the virtual interrupt flag. Pass the following information:

AX = 0902H

After the call, the carry flag is clear (this function always succeeds).

AL = 0 if virtual interrupts are disabled.
AL = 1 if virtual interrupts are enabled.

11.2.11 Vendor Specific Extensions

Some DOS extenders provide extensions to the standard set of DPMI calls. This call is used to obtain an address which must be called to use the extensions. The caller points DS:ESI to a null terminated string that specifies the vendor name or some other unique identifier to obtain the specific extension entry point.

Function 0A00H This function gets Tenberry Software's API Entry Point. Pass the following information:

AX = 0A00H
DS:ESI = Pointer to null terminated string "RATIONAL DOS/4G"

If the call succeeds, the carry flag is clear and ES:EDI = Extended API entry point. DS, FS, GS, EAX, EBX, ECX, EDX, ESI, and EBP may be modified.

If the call fails, the carry flag is set.

Notes:

1. Execute a far call to call the API entry point.
2. All extended API parameters are specified by the vendor.
3. The string comparison used to return the API entry point is case sensitive.

11.2.12 Coprocessor Status

Function 0E00H This function gets the coprocessor status. Pass the following information:

AX = 0E00H

If the call succeeds, the carry flag is clear and AX contains the coprocessor status.

Bit	*Significance*
0	MPv (MP bit in the virtual MSW/CR0). 0 = Numeric coprocessor is disabled for this client. 1 = Numeric coprocessor is disabled for this client.
1	EMv (EM bit in the virtual MSW/CR0). 0 = Client is not emulating coprocessor instructions. 1 = Client is emulating coprocessor instructions.
2	MPr (MP bit from the actual MSW/CR0). 0 = Numeric coprocessor is not present. 1 = Numeric coprocessor is present.

1	EMr (EM bit from the actual MSW/CR0). 0 = Host is not emulating coprocessor instructions. 1 = Host is emulating coprocessor instructions.
4-7	Coprocessor type. 00H = no coprocessor. 02H = 80287 03H = 80387 04H = 80486 with numeric coprocessor 05H = Pentium
8-15	Not applicable.

If the call fails, the carry flag is set.

Notes:

1. If the real EM (EMr) bit is set, the host is supplying or is capable of supplying floating-point emulation.

2. If the MPv bit is not set, the host may not need to save the coprocessor state for this virtual machine to improve system performance.

3. The MPr bit setting should be consistent with the setting of the coprocessor type information. Ignore MPr bit information if it is in conflict with the coprocessor type information.

4. If the virtual EM (EMv) bit is set, the host delivers all coprocessor exceptions to the client, and the client is performing its own floating-point emulation (wether or not a coprocessor is present or the host also has a floating-point emulator). In other words, if the EMv bit is set, the host sets the EM bit in the real CR0 while the virtual machine is active, and reflects coprocessor not present faults (int 7) to the virtual machine.

5. A client can determine the CPU type with int 31H Function 0400H, but a client should not draw any conclusions about the presence or absence of a coprocessor based on the CPU type alone.

Function 0E01H This function sets coprocessor emulation. Pass the following information:

AX = 0E01H
BX = coprocessor bits

Bit	***Significance***
0	New value of MPv bit for client's virtual CR0. 0 = Disable numeric coprocessor for this client. 1 = Enable numeric coprocessor for this client.
1	New value of EMv bit for client's virtual CR0. 0 = client will not supply coprocessor emulation. 1 = client will supply coprocessor emulation.
2-15	Not applicable.

If the call succeeds, the carry flag is clear; if it fails, the carry flag is set.

12 Utilities

This chapter describes the Tenberry Software *DOS/4GW* utility programs provided with the Open Watcom C/C++ package. Each program is described using the following format:

Purpose: This is a brief statement of what the utility program does. More specific information is provided under "Notes".

Syntax: This shows the syntax of the program. The fixed portion of each command is in a `typewriter font,` while variable parts of the command are in *italics.* Optional parts are enclosed in [brackets].

Notes: These are explanatory remarks noting major features and possible pitfalls. We explain anything special that you might need to know about the program.

See Also: This is a cross-reference to any information that is related to the program.

Example: You'll find one or more sample uses of the utility program with an explanation of what the program is doing.

Some of the utilities are *DOS/4GW*-based, protected-mode programs. To determine which programs run in protected mode and which in real, run the program. If you see the *DOS/4GW* banner, the program runs in protected mode.

12.1 DOS4GW

Purpose: Loads and executes linear executables.

Syntax: *linear_executable*

Notes: The stub program at the beginning of the linear executable invokes this program, which loads the linear executable and starts up the DOS extender. The stub program must be able to find DOS4GW: make sure it is in the path.

12.2 PMINFO

Purpose: Measures the performance of protected/real-mode switching and extended memory.

Syntax: `PMINFO.EXE`

Notes: We encourage you to distribute this program to your users.

The time-based measurements made by PMINFO may vary slightly from run to run.

Example: The following example shows the output of the PMINFO program on a 386 AT-compatible machine.

```
C>pminfo
     Protected Mode and Extended Memory Performance Measurement -- 5.00
             Copyright (c) Tenberry Software, Inc. 1987 - 1993

DOS memory   Extended memory   CPU performance equivalent to 67.0 MHz 80486
----------   ---------------
       736              8012   K bytes configured (according to BIOS).
       640             15360   K bytes physically present (SETUP).
       651              7887   K bytes available for DOS/16M programs.
22.0 (3.0)       18.9 (4.0)   MB/sec word transfer rate (wait states).
42.9 (3.0)       37.0 (4.0)   MB/sec 32-bit transfer rate (wait states).

Overall cpu and memory performance (non-floating point) for typical
DOS programs is 10.36 æ 1.04 times an 8MHz IBM PC/AT.

Protected/Real switch rate = 36156/sec (27 usec/switch, 15 up + 11 down),
DOS/16M switch mode 11 (VCPI).
```

The top information line shows that the CPU performance is equivalent to a 67.0 MHz 80486. Below are the configuration and timings for both the DOS memory and extended memory. If the computer is not equipped with extended memory, or none is available for *DOS/4GW*, the extended memory measurements may be omitted ("--").

The line "according to BIOS" shows the information provided by the BIOS (interrupts 12h and 15h function 88h). The line "SETUP", if displayed, is the configuration obtained directly from the CMOS RAM as set by the computer's setup program. It is displayed only if the numbers are different from those in the BIOS line. They will be different for computers where the BIOS has reserved memory for itself or if another program has allocated some memory and is intercepting the BIOS configuration requests to report less memory available than is physically configured. The "DOS/16M memory range", if displayed, shows the low and high addresses available to *DOS/4GW* in extended memory.

Below the configuration information is information on the memory speed *(transfer rate).* PMINFO tries to determine the memory architecture. Some architectures will perform well under some circumstances and poorly under others; PMINFO will show both the best and worst cases. The architectures detected are cache, interleaved, page-mode (or static column), and direct. Measurements are made using 32-bit accesses and reported as the number of megabytes per second that can be transferred. The number of wait states is reported in parentheses. The wait states can be a fractional number, like 0.5, if there is a wait state on writes but not on reads. Memory bandwidth (i.e., how fast the CPU can access memory) accounts for 60% to 70% of the performance for typical programs (that are not heavily dependent on floating-point math).

A performance metric developed by Tenberry Software is displayed, showing the expected throughput for the computer relative to a standard 8MHz IBM PC/AT (disk accesses and floating point are excluded). Finally, the speed with which the computer can switch between real and protected mode is displayed, both as the maximum number of round-trip switches that can occur per second, and the time for a single round-trip switch, broken out into the real-to-protected (up) and protected-to-real (down) components.

12.3 PRIVATXM

Purpose: Creates a private pool of memory for *DOS/4GW* programs.

Syntax: `PRIVATXM [-r]`

Notes: This program may be distributed to your users.

Without PRIVATXM, a *DOS/4GW* program that starts up while another *DOS/4GW* program is active uses the pool of memory built by the first program. The new program cannot change the parameters of this memory pool, so setting **DOS16M** to increase the size of the pool has no effect. To specify that the two programs use different pools of memory, use PRIVATXM.

PRIVATXM marks the active *DOS/4GW* programs as private, preventing subsequent *DOS/4GW* programs from using the same memory pool. The first *DOS/4GW* program to start after PRIVATXM sets up a new pool of memory for itself and any subsequent *DOS/4GW* programs. To release the memory used by the private programs, use the PRIVATXM `-r` option.

PRIVATXM is a TSR that requires less than 500 bytes of memory. It is not supported under DPMI.

Example: The following example creates a 512KB memory pool that is shared by two *DOS/4GW* TSRs. Subsequent *DOS/4GW* programs use a different memory pool.

C>set DOS16M= :512	Specifies the size of the memory pool.
C>TSR1	Sets up the memory pool at startup.
C>TSR2	This TSR shares the pool built by TSR1.
C>PRIVATXM	Makes subsequent *DOS/4GW* programs use a new memory pool.
C>set DOS16M=	Specifies an unlimited size for the new pool.
C>PROGRAM3	This program uses the new memory pool.
C>PRIVATXM -R	Releases the 512KB memory pool used by the TSRs. (If the TSRs shut down, their memory is not released unless PRIVATXM is released.)

12.4 RMINFO

Purpose: Supplies configuration information and the basis for real/protected-mode switching in your machine.

Syntax: `RMINFO.EXE`

Notes: This program may be distributed to your users.

RMINFO starts up *DOS/4GW*, but stops your machine just short of switching from real mode to protected mode and displays configuration information about your computer. The information shown by RMINFO can help determine why *DOS/4GW* applications won't run on a particular machine. Run RMINFO if PMINFO does not run to completion.

Example: The following example shows the output of the RMINFO program on an 386 AT-compatible machine.

```
C>rminfo

DOS/16M Real Mode Information Program 5.00
Copyright (C) Tenberry Software, Inc. 1987 - 1993

Machine and Environment:
        Processor:              i386, coprocessor present
        Machine type:           10 (AT-compatible)
        A20 now:                enabled
        A20 switch rigor:       disabled
        DPMI host found
Switching Functions:
        To PM switch:           DPMI
        To RM switch:           DPMI
        Nominal switch mode:    0
        Switch control flags:   0000
Memory Interfaces:
        DPMI may provide:       16384K returnable
        Contiguous DOS memory:  463K
```

The information provided by RMINFO includes:

Machine and Environment:

Processor: processor type, coprocessor present/not present

Machine type:

(NEC 9801)
(PS/2-compatible)
(AT-compatible)
(FM R)
(AT&T 6300+)
(AT-compatible)
(C&T 230 chipset)
(AT-compatible)
(AT-compatible)
(Acer)
(Zenith)
(Hitachi)
(Okidata)
(PS/55)

A20 now: Current state of Address line 20.

A20 switch rigor: Whether DOS4GW rigorously controls enabling and disabling of Address line 20 when switching modes.

PS feature flag

XMS host found Whether your system has any software using extended memory under the XMS discipline.

VCPI host found Whether your system has any software using extended memory under the VCPI discipline.

page table 0 at: x000h

DPMI host found

DOS/16M resident with private/public memory

Switching Functions:

A20 switching:

To PM switch: reset catch:
pre-PM prep:
post-PM-switch:

To RM switch:
pre-RM prep:
reset method:
post-reset:
reset uncatch:

Nominal switch mode: x

Switch control flags: xxxxh

Memory Interfaces:

(VCPI remapping in effect)

DPMI may provide: xxxxxK returnable

VCPI may provide: xxxxxK returnable

Top-down

Other16M

Forced

Contiguous DOS memory:

13 Error Messages

The following lists DOS/4G error messages, with descriptions of the circumstances in which the error is most likely to occur, and suggestions for remedying the problem. Some error messages pertaining to features — like DLLs — that are not supported in *DOS/4GW* will not arise with that product. In the following descriptions, references to DOS/4G, DOS4G, or DOS4G.EXE may be replaced by DOS/4GW, DOS4GW, or DOS4GW.EXE should the error message arise when using *DOS/4GW*.

13.1 Kernel Error Messages

This section describes error messages from the DOS/16M kernel embedded in DOS/4G. Kernel error messages may occur because of severe resource shortages, corruption of DOS4GW.EXE, corruption of memory, operating system incompatibilities, or internal errors in DOS/4GW. All of these messages are quite rare.

0. involuntary switch to real mode

The computer was in protected mode but switched to real mode without going through DOS/16M. This error most often occurs because of an unrecoverable stack segment exception (stack overflow), but can also occur if the Global Descriptor Table or Interrupt Descriptor Table is corrupted. Increase the stack size, recompile your program with stack overflow checking, or look into ways that the descriptor tables may have been overwritten.

1. not enough extended memory

2. not a DOS/16M executable <filename>

DOS4G.EXE, or a bound DOS/4G application, has probably been corrupted in some way. Rebuild or recopy the file.

3. no DOS memory for transparent segment

4. cannot make transparent segment

5. too many transparent segments

6. not enough memory to load program

There is not enough memory to load DOS/4G. Make more memory available and try again.

7. no relocation segment

8. cannot open file <filename>

The DOS/16M loader cannot load DOS/4G, probably because DOS has run out of file units. Set a larger FILES= entry in CONFIG.SYS, reboot, and try again.

9. cannot allocate tstack

There is not enough memory to load DOS/4G. Make more memory available and try again.

10. cannot allocate memory for GDT

There is not enough memory to load DOS/4G. Make more memory available and try again.

11. no passup stack selectors -- GDT too small

This error indicates an internal error in DOS/4G or an incompatibility with other software.

12. no control program selectors -- GDT too small

This error indicates an internal error in DOS/4G or an incompatibility with other software.

13. cannot allocate transfer buffer

There is not enough memory to load DOS/4G. Make more memory available and try again.

14. premature EOF

DOS4G.EXE, or a bound DOS/4G application, has probably been corrupted in some way. Rebuild or recopy the file.

15. protected mode available only with 386 or 486

DOS/4G requires an 80386 (or later) CPU. It cannot run on an 80286 or earlier CPU.

16. cannot run under OS/2

17. system software does not follow VCPI or DPMI specifications

Some memory resident program has put your 386 or 486 CPU into Virtual 8086 mode. This is done to provide special memory services to DOS programs, such as EMS simulation (EMS interface without EMS hardware) or high memory. In this mode, it is not possible to switch into protected mode unless the resident software follows a standard that DOS/16M supports (DPMI, VCPI, and XMS are the most common). Contact the vendor of your memory management software.

18. you must specify an extended memory range (SET DOS16M=)

On some Japanese machines that are not IBM AT-compatible, and have no protocol for managing extended memory, you must set the DOS16M environment variable to specify the range of available extended memory.

19. computer must be AT- or PS/2- compatible

20. unsupported DOS16M switchmode choice

21. requires DOS 3.0 or later

22. cannot free memory

This error probably indicates that memory was corrupted during execution of your program.

23. no memory for VCPI page table

There is not enough memory to load DOS/4G. Make more memory available and try again.

24. VCPI page table address incorrect

This is an internal error.

25. cannot initialize VCPI

This error indicates an incompatibility with other software. DOS/16M has detected that VCPI is present, but VCPI returns an error when DOS/16M tries to initialize the interface.

26. 8042 timeout

27. extended memory is configured but it cannot be allocated

28. memory error, avail loop

This error probably indicates that memory was corrupted during execution of your program. Using an invalid or stale alias selector may cause this error. Incorrect manipulation of segment descriptors may also cause it.

29. memory error, out of range

This error probably indicates that memory was corrupted during execution of your program. Writing through an invalid or stale alias selector may cause this error.

30. program must be built -AUTO for DPMI

31. protected mode already in use in this DPMI virtual machine

32. DPMI host error (possibly insufficient memory)

33. DPMI host error (need 64K XMS)

34. DPMI host error (cannot lock stack)

Any of these errors (32, 33, 34) probably indicate insufficient memory under DPMI. Under Windows, you might try making more physical memory available by eliminating or reducing any RAM drives or disk caches. You might also try editing DEFAULT.PIF so that at least 64KB of XMS memory is available to non-Windows programs. Under OS/2, you want to increase the DPMI_MEMORY_LIMIT in the DOS box settings.

35. General Protection Fault

This message probably indicates an internal error in DOS/4G. Faults generated by your program should cause error 2001 instead.

36. The DOS16M.386 virtual device driver was never loaded

37. Unable to reserve selectors for DOS16M.386 Windows driver

38. Cannot use extended memory: HIMEM.SYS not version 2

This error indicates an incompatibility with an old version of HIMEM.SYS.

39. An obsolete version of DOS16M.386 was loaded

40. not enough available extended memory (XMIN)

This message probably indicates an incompatibility with your memory manager or its configuration. Try configuring the memory manager to provide more extended memory, or change memory managers.

13.2 DOS/4G Errors

1000 "can't hook interrupts"

A DPMI host has prevented DOS/4G from loading. Please contact Tenberry Technical Support.

1001 "error in interrupt chain"

DOS/4G internal error. Please contact Tenberry Technical Support.

1003 "can't lock extender kernel in memory"

DOS/4G couldn't lock the kernel in physical memory, probably because of a memory shortage.

1004 "syntax is DOS4G <executable.xxx>"

You must specify a program name.

1005 "not enough memory for dispatcher data"

There is not enough memory for DOS/4G to manage user-installed interrupt handlers properly. Free some memory for the DOS/4G application.

1007 "can't find file <program> to load"

DOS/4G could not open the specified program. Probably the file didn't exist. It is possible that DOS ran out of file handles, or that a network or similar utility has prohibited read access to the program. Make sure that the file name was spelled correctly.

1008 "can't load executable format for file <filename> [<error code>]"

DOS/4G did not recognize the specified file as a valid executable file. DOS/4G can load linear executables (LE and LX) and EXPs (BW). The error code is for Tenberry Software's use.

1009 "program <filename> is not bound"

This message does not occur in DOS/4G, only DOS/4GW Professional; the latter requires that the DOS extender be bound to the program file. The error signals an attempt to load

1010 "can't initialize loader <loader> [<error code>]"

DOS/4G could not initialize the named loader, probably because of a resource shortage. Try making more memory available. If that doesn't work, please contact Tenberry Technical Support. The error code is for Tenberry Software' use.

1011 "VMM initialization error [<error code>]"

DOS/4G could not initialize the Virtual Memory Manager, probably because of a resource shortage. Try making more memory available. If that doesn't work, please contact Tenberry Technical Support. The error code is for Tenberry Software' use.

1012 "<filename> is not a WATCOM program"

This message does not occur in DOS/4G, only DOS/4GW and DOS/4GW Professional. Those extenders only support WATCOM 32-bit compilers.

1013 "int 31h initialization error"

DOS/4G was unable to initialize the code that handles Interrupt 31h, probably because of an internal error. Please call Tenberry Technical Support.

1100 "assertion \"<statement>\" failed (<file>:<line>)"

DOS/4G internal error. Please contact Tenberry Technical Support.

1200 "invalid EXP executable format"

DOS/4G tried to load an EXP, but couldn't. The executable file is probably corrupted.

1201 "program must be built -AUTO for DPMI"

Under DPMI, DOS/4G can only load EXPs that have been linked with the GLU -AUTO or -DPMI switch.

1202 "can't allocate memory for GDT"

There is not enough memory available for DOS/4G to build a Global Descriptor Table. Make more memory available.

1203 "premature EOF"

DOS/4G tried to load an EXP but couldn't. The file is probably corrupted.

1204 "not enough memory to load program"

There is not enough memory available for DOS/4G to load your program. Make more memory available.

1301 "invalid linear executable format"

DOS/4G cannot recognize the program file as a LINEXE format. Make sure that you specified the correct file name.

1304 "file I/O seek error"

DOS/4G was unable to seek to a file location that should exist. This usually indicates truncated program files or problems with the storage device from which your program loads. Run CHKDSK or a similar utility to begin determining possible causes.

1305 "file I/O read error"

DOS/4G was unable to read a file location that should contain program data. This usually indicates truncated program files or problems with the storage device from which your program loads. Run CHKDSK or a similar utility to begin determining possible causes.

1307 "not enough memory"

As it attempted to load your program, DOS/4G ran out of memory. Make more memory available, or enable VMM.

1308 "can't load requested program"

1309 "can't load requested program"

1311 "can't load requested program"

1312 "can't load requested program"

DOS/4G cannot load your program for some reason. Contact Tenberry Technical Support.

1313 "can't resolve external references"

DOS/4G was unable to resolve all references to DLLs for the requested program, or the program contained unsupported fixup types. Use EXEHDR or a similar LINEXE dump utility to see what references your program makes and what special fixup records might be present.

1314 "not enough lockable memory"

As it attempted to load your program, DOS/4G encountered a refusal to lock a virtual memory region. Some memory must be locked in order to handle demand-load page faults. Make more physical memory available.

1315 "can't load requested program"

1316 "can't load requested program"

DOS/4G cannot load your program for some reason. Contact Tenberry Technical Support.

1317 "program has no stack"

DOS/4G reports this error when you try to run a program with no stack. Rebuild your program, building in a stack.

2000 "deinitializing twice"

DOS/4G internal error. Please contact Tenberry Technical Support.

2001 "exception <exception_number> (<exception_description>) at <selector:offset>"

Your program has generated an exception. For information about interpreting this message, see the file COMMON.DOC.

2002 "transfer stack overflow at <selector:offset>"

Your program has overflowed the DOS/4G transfer stack. For information about interpreting this message, see the file COMMON.DOC.

2300 " can't find <DLL>.<ordinal> - referenced from <module>"

DOS/4G could not find the ordinal listed in the specified DLL, or it could not find the DLL at all. Correct or remove the reference, and make sure that DOS/4G can find the DLL.

DOS/4G looks for DLLs in the following directories:

- The directory specified by the Libpath32 configuration option (which defaults to the directory of the main application file).
- The directory or directories specified by the LIBPATH32 environment variable.
- Directories specified in the PATH.

2301 "can't find <DLL>.<name> - referenced from <module>"

DOS/4G could not find the entry point named in the specified module. Correct or remove the reference, and make sure that DOS/4G can find the DLL.

2302 "DLL modules not supported"

This DOS/4GW Professional error message arises when an application references or tries to explicitly load a DLL. DOS/4GW Professional does not support DLLs.

2303 "internal LINEXE object limit reached"

DOS/4G currently handles a maximum of 128 LINEXE objects, including all .DLL and .EXE files. Most .EXE or .DLL files use only three or four objects. If possible, reduce the number of objects, or contact Tenberry Technical Support.

2500 "can't connect to extender kernel"

DOS/4G internal error. Please contact Tenberry Technical Support.

2503 "not enough disk space for swapping - <count> byes required"

VMM was unable to create a swap file of the required size. Increase the amount of disk space available.

2504 "can't create swap file \<filename>\""

VMM was unable to create the swap file. This could be because the swap file is specified for a nonexistent drive or on a drive that is read-only. Set the SWAPNAME parameter to change the location of the swap file.

2505 "not enough memory for <table>"

VMM was unable to get sufficient extended memory for internal tables. Make more memory available. If <table> is page buffer, make more DOS memory available.

2506 "not enough physical memory (minmem)"

There is less physical memory available than the amount specified by the MINMEM parameter. Make more memory available.

2511 "swap out error [<error code>]"

Unknown disk error. The error code is for Tenberry Software' use.

2512 "swap in error [<error code>]"

Unknown disk error. The error code is for Tenberry Software' use.

2514 "can't open trace file"

VMM could not open the VMM.TRC file in the current directory for writing. If the directory already has a VMM.TRC file, delete it. If not, there may not be enough memory on the drive for the trace file, or DOS may not have any more file handles.

2520 "can't hook int 31h"

DOS/4G internal error. Please contact Tenberry Technical Support.

2523 "page fault on non-present mapped page"

Your program references memory that has been mapped to a nonexistent physical device, using DPMI function 508h. Make sure the device is present, or remove the reference.

2524 "page fault on uncommitted page"

Your program references memory reserved with a call to DPMI function

504h, but never committed (using a DPMI 507h or 508h call). Commit the memory before you reference it.

3301 "unhandled EMPTYFWD, GATE16, or unknown relocation"

3302 "unhandled ALIAS16 reference to unaliased object"

3304 "unhandled or unknown relocation"

If your program was built for another platform that supports the LINEXE format, it may contain a construct that DOS/4G does not currently support, such as a call gate. This message may also occur if your program has a problem mixing 16- and 32-bit code. A linker error is another likely cause.

14 DOS/4GW Commonly Asked Questions

The following information has been provided by Tenberry Software, Inc. for their DOS/4GW and DOS/4GW Professional product. The content of this chapter has been edited by Open Watcom. In most cases, the information is applicable to both products.

This chapter covers the following topics:

- Access to technical support
- Differences within the DOS/4G product line
- Addressing
- Interrupt and exception handling
- Memory management
- DOS, BIOS, and mouse services
- Virtual memory
- Debugging
- Compatibility

14.1 Access to Technical Support

1a. How to reach technical support.

Here are the various ways you may contact Tenberry Software for technical support.

```
WWW:    http://www.tenberry.com/dos4g/
Email:  4gwhelp@tenberry.com
Phone:  1.480.767.8868
Fax:    1.480.767.8709
Mail:   Tenberry Software, Inc.
        PO Box 20050
        Fountain Hills, Arizona
        U.S.A  85269-0050
```

PLEASE GIVE YOUR SERIAL NUMBER WHEN YOU CONTACT TENBERRY.

1b. When to contact Open Watcom, when to contact Tenberry.

Since DOS/4GW Professional is intended to be completely compatible with DOS/4GW, you may wish to ascertain whether your program works properly under DOS/4GW before contacting Tenberry Software for technical support. (This is likely to be the second question we ask you, after your serial number.)

If your program fails under both DOS/4GW and DOS/4GW Professional, and you suspect your own code or a problem compiling or linking, you may wish to contact Open Watcom first. Tenberry Software support personnel are not able to help you with most programming questions, or questions about using the Open Watcom tools.

If your program only fails with DOS/4GW Professional, you have probably found a bug in DOS/4GW Professional, so please contact us right away.

1c. Telephone support.

Tenberry Software's hours for telephone support are 9am-6pm EST. Please note that telephone support is free for the first 30 days only. A one-year contract for continuing telephone support on DOS/4GW Professional is US$500 per developer, including an update subscription for one year, to customers in the United States and Canada; for overseas customers, the price is $600. Site licenses may be negotiated.

There is no time limit on free support by fax, mail, or electronic means.

1d. References.

The DOS/4GW documentation from Open Watcom is the primary reference for DOS/4GW Professional as well. Another useful reference is the DPMI specification. In the past, the DPMI specification could be obtained free of charge by contacting Intel Literature. We have been advised that the DPMI specification is no longer available in printed form.

However, the DPMI 1.0 specification can be obtained at:

```
http://www.delorie.com/djgpp/doc/dpmi/
```

Online HTML as well as a downloadable archive are provided.

14.2 Differences Within the DOS/4G Product Line

2a. DOS/4GW Professional versus DOS/4GW

DOS/4GW Professional was designed to be a higher-performance version of DOS/4GW suitable for commercial applications. Here is a summary of the advantages of DOS/4GW Professional with respect to DOS/4GW:

- Extender binds to the application program file
- Extender startup time has been reduced
- Support for Open Watcom floating-point emulator has been optimized
- Virtual memory manager performance has been greatly improved

- Under VMM, programs are demand loaded
- Virtual address space is 4 GB instead of 32 MB
- Extender memory requirements have been reduced by more than 50K
- Extender disk space requirements have been reduced by 40K
- Can omit virtual memory manager to save 50K more disk space
- Support for INT 31h functions 301h-304h and 702h-703h

DOS/4GW Professional is intended to be fully compatible with programs written for DOS/4GW 1.9 and up. The only functional difference is that the extender is bound to your program instead of residing in a separate file. Not only does this help reduce startup time, but it eliminates version-control problems when someone has both DOS/4GW and DOS/4GW Professional applications present on one machine.

2b. DOS/4GW Professional versus DOS/4G.

DOS/4GW Professional is not intended to provide any other new DOS extender functionality. Tenberry Software's top-of-the-line 32-bit extender, DOS/4G, is not sold on a retail basis but is of special interest to developers who require more flexibility (such as OEMs). DOS/4G offers these additional features beyond DOS/4GW and DOS/4GW Professional:

- Complete documentation
- DLL support
- TSR support
- Support for INT 31h functions 301h-306h, 504h-50Ah, 702h-703h
- A C language API that offers more control over interrupt handling and program loading, as well as making it easier to use the extender
- An optional (more protected) nonzero-based flat memory model
- Remappable error messages
- More configuration options
- The D32 debugger, GLU linker, and other tools
- Support for other compilers besides Open Watcom
- A higher level of technical support
- Custom work is available (e.g., support for additional executable formats, operating system API emulations, mixed 16-bit and 32-bit code)

Please contact Tenberry Software if you have questions about other products (present or future) in the DOS/4G line.

2c. DPMI functions supported by DOS/4GW.

Note that when a DOS/4GW application runs under a DPMI host, such as Windows 3.1 in enhanced mode, an OS/2 virtual DOS machine, 386Max (with DEBUG=DPMIXCOPY), or QDPMI (with EXTCHKOFF), the DPMI host provides the DPMI services, not DOS/4GW. The DPMI host also provides virtual memory, if any. Performance (speed and memory use) under different DPMI hosts varies greatly due to the quality of the DPMI implementation.

These are the services provided by DOS/4GW and DOS/4GW Professional in the absence of a DPMI host.

0000	Allocate LDT Descriptors
0001	Free LDT Descriptor
0002	Map Real-Mode Segment to Descriptor
0003	Get Selector Increment Value
0006	Get Segment Base Address
0007	Set Segment Base Address
0008	Set Segment Limit
0009	Set Descriptor Access Rights
000A	Create Alias Descriptor
000B	Get Descriptor
000C	Set Descriptor
000D	Allocate Specific LDT Descriptor
0100	Allocate DOS Memory Block
0101	Free DOS Memory Block
0102	Resize DOS Memory Block
0200	Get Real-Mode Interrupt Vector
0201	Set Real-Mode Interrupt Vector
0202	Get Processor Exception Handler
0203	Set Processor Exception Handler
0204	Get Protected-Mode Interrupt Vector
0205	Set Protected-Mode Interrupt Vector
0300	Simulate Real-Mode Interrupt
0301	Call Real-Mode Procedure with Far Return Frame (DOS/4GW Professional only)
0302	Call Real-Mode Procedure with IRET Frame (DOS/4GW Professional only)
0303	Allocate Real-Mode Callback Address (DOS/4GW Professional only)
0304	Free Real-Mode Callback Address (DOS/4GW Professional only)
0400	Get DPMI Version
0500	Get Free Memory Information
0501	Allocate Memory Block
0502	Free Memory Block
0503	Resize Memory Block
0600	Lock Linear Region
0601	Unlock Linear Region
0604	Get Page Size (VM only)
0702	Mark Page as Demand Paging Candidate (DOS/4GW Professional only)

0703	Discard Page Contents (DOS/4GW Professional only)
0800	Physical Address Mapping
0801	Free Physical Address Mapping
0900	Get and Disable Virtual Interrupt State
0901	Get and Enable Virtual Interrupt State
0902	Get Virtual Interrupt State
0A00	Get Tenberry Software API Entry Point
0E00	Get Coprocessor Status
0E01	Set Coprocessor Emulation

14.3 Addressing

3a. Converting between pointers and linear addresses.

Because DOS/4GW uses a zero-based flat memory model, converting between pointers and linear addresses is trivial. A pointer value is always relative to the current segment (the value in CS for a code pointer, or in DS or SS for a data pointer). The segment bases for the default DS, SS, and CS are all zero. Hence a near pointer is exactly the same thing as a linear address: a null pointer points to linear address 0, and a pointer with value 0x10000 points to linear address 0x10000.

3b. Converting between code and data pointers.

Because DS and CS have the same base address, they are natural aliases for each other. To create a data alias for a code pointer, merely create a data pointer and set it equal to the code pointer. It's not necessary for you to create your own alias descriptor. Similarly, to create a code alias for a data pointer, merely create a code pointer and set it equal to the data pointer.

3c. Converting between pointers and low memory addresses.

Linear addresses under 1 MB map directly to physical memory. Hence the real-mode interrupt vector table is at address 0, the BIOS data segment is at address 0x400, the monochrome video memory is at address 0xB0000, and the color video memory is at address 0xB8000. To read and write any of these, you can just use a pointer set to the proper address. You don't need to create a far pointer, using some magic segment value.

3d. Converting between linear and physical addresses.

Linear addresses at or above 1 MB do not map directly to physical memory, so you can not in general read or write extended memory directly, nor can you tell how a particular block of extended memory has been used.

DOS/4GW supports the DPMI call INT 31h/800h, which maps physical addresses to linear addresses. In other words, if you have a peripheral device in your machine that has memory at a physical address of 256 MB, you can issue this call to create a linear address that points to that physical memory. The linear address is the same thing as a near pointer to the memory and can be manipulated as such.

There is no way in a DPMI environment to determine the physical address corresponding to a given linear address. This is part of the design of DPMI. You must design your application accordingly.

3e. Null pointer checking.

DOS/4GW will trap references to the first sixteen bytes of physical memory if you set the environment variable DOS4G=NULLP. This is currently the only null-pointer check facility provided by DOS/4GW.

As of release 1.95, DOS/4GW traps both reads and writes. Prior to this, it only trapped writes.

You may experience problems if you set DOS4G=NULLP and use some versions of the Open Watcom Debugger with a 1.95 or later extender. These problems have been corrected in later versions of the Open Watcom Debugger.

14.4 Interrupt and Exception Handling

4a. Handling asynchronous interrupts.

Under DOS/4GW, there is a convenient way to handle asynchronous interrupts and an efficient way to handle them.

Because your CPU may be in either protected mode (when 32-bit code is executing) or real mode (a DOS or BIOS call) when a hardware interrupt comes in, you have to be prepared to handle interrupts in either mode. Otherwise, you may miss interrupts.

You can handle both real-mode and protected-mode interrupts with a single handler, if 1) the interrupt is in the auto-passup range, 8 to 2Eh; and 2) you install a handler with INT 21h/25h or _dos_setvect(); 3) you do not install a handler for the same interrupt using any other mechanism. DOS/4GW will route both protected-mode interrupts and real-mode interrupts to your protected-mode handler. This is the convenient way.

The efficient way is to install separate real-mode and protected-mode handlers for your interrupt, so your CPU won't need to do unnecessary mode switches. Writing a real-mode handler is tricky; all you can reasonably expect to do is save data in a buffer and IRET. Your protected-mode code can periodically check the buffer and process any queued data. (Remember, protected-mode code can access data and execute code in low memory, but real-mode code can't access data or execute code in extended memory.)

For performance, it doesn't matter how you install the real-mode handler, but we recommend the DPMI function INT 31h/201h for portability.

It does matter how you install the protected-mode handler. You can't install it directly into the IDT, because a DPMI provider must distinguish between interrupts and exceptions and maintain separate handler chains. Installing with INT 31h/205h is the recommended way to install your protected-mode handler for both performance and portability.

If you install a protected-mode handler with INT 21h/25h, both interrupts and exceptions will be funneled to your handler, to mimic DOS. Since DPMI exception handlers and interrupt handlers are called with different stack frames, DOS/4GW executes a layer of code to cover these differences up; the same layer is used to support the DOS/4G API (not part of DOS/4GW). This layer is the reason that hooking with INT 21h/25h is less efficient than hooking with INT 31h/205h.

4b. Handling asynchronous interrupts in the second IRQ range.

Because the second IRQ range (normally INTs 70h-77h) is outside the DOS/4GW auto-passup range (8-2Eh, excluding 21h) you may not handle these interrupts with a single handler, as described above (the "convenient" method). You must install separate real-mode and protected-mode handlers (the "efficient" method).

DOS/4G does allow you to specify additional passup interrupts, however.

4c. Asynchronous interrupt handlers and DPMI.

The DPMI specification requires that all code and data referenced by a hardware interrupt handler MUST be locked at interrupt time. A DPMI virtual memory manager can use the DOS file system to swap pages of memory to and from the disk; because DOS is not reentrant, a DPMI host is not required to be able to handle page faults during asynchronous interrupts. Use INT 31h/600h (Lock Linear Region) to lock an address range in memory.

If you fail to lock all of your code and data, your program may run under DOS/4GW, but fail under the DOS/4GW Virtual Memory Manager or under another DPMI host such as Windows or OS/2.

You should also lock the code and data of a mouse callback function.

4d. Open Watcom signal() function and Ctrl-Break.

In earlier versions of the Open Watcom C/C++ library, there was a bug that caused signal(SIGBREAK) not to work. Calling signal(SIGBREAK) did not actually install an interrupt handler for Ctrl-Break (INT 1Bh), so Ctrl-Break would terminate the application rather than invoking the signal handler.

With these earlier versions of the library, you could work around this problem by hooking INT 1Bh directly. With release 10.0, this problem has been fixed.

4e. More tips on writing hardware interrupt handlers.

- It's more like handling interrupts in real mode than not.

  The same problems arise when writing hardware interrupt handlers for protected mode as arise for real mode. We assume you know how to write real-mode handlers; if our suggestions don't seem clear, you might want to brush up on real-mode interrupt programming.

- Minimize the amount of time spent in your interrupt handlers.

  When your interrupt handlers are called, interrupts are disabled. This means that no other system tasks can be performed until you enable interrupts (an STI instruction) or until your handler returns. In general, it's a good idea to handle interrupts as quickly as possible.

- Minimize the amount of time spent in the DOS extender by installing separate real-mode and protected-mode handlers.

  If you use a passup interrupt handler, so that interrupts received in real mode are resignalled in protected mode by the extender, your application has to switch from real mode to protected mode to real mode once per interrupt. Mode switching is a time-consuming process, and interrupts are disabled during a mode switch. Therefore, if you're concerned about performance, you should install separate handlers for real-mode and protected-mode interrupts, eliminating the mode switch.

• If you can't just set a flag and return, enable interrupts (STI).

Handlers that do more than just set a flag or store data in a buffer should re-enable interrupts as soon as it's safe to do so. In other words, save your registers on the stack, establish your addressing conventions, switch stacks if you're going to — and then enable interrupts (STI), to give priority to other hardware interrupts.

• If you enable interrupts (STI), you should disable interrupts (CLI).

Because some DPMI hosts virtualize the interrupt flag, if you do an STI in your handler, you should be sure to do a CLI before you return. (CLI, then switch back to the original stack if you switched away, then restore registers, then IRET.) If you don't do this, the IRET will not necessarily restore the previous interrupt flag state, and your program may crash. This is a difference from real-mode programming, and it tends to show up as a problem when you try running your program in a Windows or OS/2 DOS box for the first time (but not before).

• Add a reentrancy check.

If your handler doesn't complete its work by the time the next interrupt is signalled, then interrupts can quickly nest to the point of overflowing the transfer stack. This is a design flaw in your program, not in the DOS extender; a real-mode DOS program can have exactly the same behavior. If you can conceive of a situation where your interrupt handler can be called again before the first instance returns, you need to code in a reentrancy check of some sort (before you switch stacks and enable interrupts (STI), obviously).

Remember that interrupts can take different amounts of time to execute on different machines; the CPU manufacturer, CPU speed, speed of memory accesses, and CMOS settings (e.g. "system BIOS shadowing") can all affect performance in subtle ways. We recommend you program defensively and always check for unexpected reentry, to avoid transfer stack overflows.

• Switch to your own stack.

Interrupt handlers are called on a stack that typically has only a small amount of stack available (512 bytes or less). If you need to use more stack than this, you have to switch to your own stack on entry into the handler, and switch back before returning.

If you want to use C run-time library functions, which are compiled for flat memory model (SS == DS, and the base of CS == the base of DS), you need to switch back to a stack in the flat data segment first.

Note that switching stacks by itself won't prevent transfer stack overflows of the kind described above.

14.5 Memory Management

5a. Using the realloc() function.

In versions of Open Watcom C/C++ prior to 9.5b, there was a bug in the library implementation of realloc() under DOS/4GW and DOS/4GW Professional. This bug was corrected by Open Watcom in the 9.5b release.

5b. Using all of physical memory.

DOS/4GW Professional is currently limited to 64 MB of physical memory. We do not expect to be able to fix this problem for at least six months. If you need more than 64 MB of memory, you must use virtual memory.

14.6 DOS, BIOS, and Mouse Services

6a. Speeding up file I/O.

The best way to speed up DOS file I/O in DOS/4GW is to write large blocks (up to 65535 bytes, or the largest number that will fit in a 16-bit int) at a time from a buffer in low memory. In this case, DOS/4GW has to copy the least amount of data and make the fewest number of DOS calls in order to process the I/O request.

Low memory is allocated through INT 31h/0100h, Allocate DOS Memory Block. You can convert the real-mode segment address returned by INT 31h/0100h to a pointer (suitable for passing to setvbuf()) by shifting it left four bits.

6b. Spawning.

It is possible to spawn one DOS/4GW application from another. However, two copies of the DOS extender will be loaded into memory. DOS/4G supports loading of multiple programs atop a single extender, as well as DLLs.

6c. Mouse callbacks.

DOS/4GW Professional now supports the INT 31h interface for managing real-mode callbacks. However, you don't need to bother with them for their single most important application: mouse callback functions. Just register your protected-mode mouse callback function as you would in real mode, by issuing INT 33h/0Ch with the event mask in CX and the function address in ES:EDX, and your function will work as expected.

Because a mouse callback function is called asynchronously, the same locking requirement exists for a mouse callback function as for a hardware interrupt handler. See (4c) above.

6d. VESA support.

While DOS/4GW automatically handles most INT 10h functions so that you can you can issue them from protected mode, it does not translate the INT 10h VESA extensions. The workaround is to use INT 31h/300h (Simulate Real-Mode Interrupt).

14.7 Virtual Memory

7a. Testing for the presence of VMM.

INT 31h/400h returns a value (BX, bit 2) that tells if virtual memory is available. Under a DPMI host such as Windows 3.1, this will be the host's virtual memory manager, not DOS/4GW's.

You can test for the presence of a DOS/4G-family DOS extender with INT 31h/0A00h, with a pointer to the null-terminated string "RATIONAL DOS/4G" in DS:ESI. If the function returns with carry clear, a DOS/4G-family extender is running.

7b. Reserving memory for a spawned application.

If you spawn one DOS/4GW application from another, you should set the DELETESWAP configuration option (i.e., SET DOS4GVM=deleteswap) so that the two applications don't try to use the same swap file. You should also set the MAXMEM option low enough so that the parent application doesn't take all available physical memory; memory that's been reserved by the parent application is not available to the child application.

7c. Instability under VMM.

A program that hooks hardware interrupts, and works fine without VMM but crashes sporadically with it, probably needs to lock the code and data for its hardware interrupt handlers down in memory. DOS/4GW does not support page faults during hardware interrupts, because DOS services may not be available at that time. See (4c) and (6c) above.

Memory can be locked down with INT 31h/600h (Lock Linear Region).

7d. Running out of memory with a huge virtual address space.

Because DOS/4GW has to create page tables to describe your virtual address space, we recommend that you set your VIRTUALSIZE parameter just large enough to accommodate your program. If you set your VIRTUALSIZE to 4 GB, the physical memory occupied by the page tables will be 4 MB, and that memory will not be available to DOS/4GW.

7e. Reducing the size of the swap file.

DOS/4GW will normally create a swap file equal to your VIRTUALSIZE setting, for efficiency. However, if you set the SWAPMIN parameter to a size (in KB), DOS/4GW will start with a swap file of that size, and will grow the swap file when it has to. The SWAPINC value (default 64 KB) controls the incremental size by which the swap file will grow.

7f. Deleting the swap file.

The DELETESWAP option has two effects: telling DOS/4GW to delete the swap file when it exits, and causing DOS/4GW to provide a unique swap file name if an explicit SWAPNAME setting was not given.

DELETESWAP is required if one DOS/4GW application is to spawn another; see (7b) above.

7g. Improving demand-load performance of large static arrays.

DOS/4GW demand-loading feature normally cuts the load time of a large program drastically. However, if your program has large amounts of global, zero-initialized data (storage class BSS), the Open Watcom startup code will explicitly zero it (version 9.5a or earlier). Because the zeroing operation touches every page of the data, the benefits of demand-loading are lost.

Demand loading can be made fast again by taking advantage of the fact that DOS/4GW automatically zeroes pages of BSS data as they are loaded. You can make this change yourself by inserting a few lines into the startup routine, assembling it (MASM 6.0 will work), and listing the modified object module first when you link your program.

Here are the changes for `\watcom\src\startup\386\cstart3r.asm` (startup module from the C/C++ 9.5 compiler, library using register calling conventions). You can modify the workaround easily for other Open Watcom compilers:

```
        ...                     ; cstart3r.asm, circa line 332
                                ; end of _BSS segment (start of STACK)
        mov     ecx,offset DGROUP:_end
                                ; start of _BSS segment
        mov     edi,offset DGROUP:_edata
;-------------------------------; RSI OPTIMIZATION
        mov     eax, edi        ; minimize _BSS initialization loop
        or      eax, 0FFFh      ; compute address of first page after
        inc     eax             ;   start of _BSS
        cmp     eax, ecx        ; if _BSS extends onto that page,
        jae     allzero         ;   then we can rely on the loader
        mov     ecx, eax        ;   zeroing the remaining pages
allzero:                        ;
;-------------------------------; END RSI OPTIMIZATION
        sub     ecx,edi         ; calc # of bytes in _BSS segment
        mov     dl,cl           ; save bottom 2 bits of count in edx
        shr     ecx,2           ; calc # of dwords
        sub     eax,eax         ; zero the _BSS segment
        rep     stosd           ; ...
        mov     cl,dl           ; get bottom 2 bits of count
        and     cl,3            ; ...
        rep     stosb           ; ...
        ...
```

Note that the 9.5b and later versions of the Open Watcom C library already contain this enhancement.

7h. How should I configure VM for best performance?

Here are some recommendations for setting up the DOS/4GW virtual memory manager.

VIRTUALSIZE Set to no more than twice the total amount of memory (virtual and otherwise) your program requires. If your program has 16 MB of code and data, set to 32 MB. (There is only a small penalty for setting this value larger than you will need, but your program won't run if you set it too low.) See (7d) above.

MINMEM Set to the minimum hardware requirement for running your application. (If you require a 2 MB machine, set to 2048).

MAXMEM Set to the maximum amount of memory you want your application to use. If you don't spawn any other applications, set this large (e.g., 32000) to make sure you can use all available physical memory. If you do spawn, see (7b) above.

SWAPMIN Don't use this if you want the best possible VM performance. The trade-off is that DOS/4GW will create a swap file as big as your VIRTUALSIZE.

SWAPINC Don't use this if you want the best possible VM performance.

DELETESWAP DOS/4GW's VM will start up slightly slower if it has to create the swap file afresh each time. However, unless your swap file is very large, DELETESWAP is a reasonable choice; it may be required if you spawn another DOS/4GW program at the same time. See (7b) above.

14.8 Debugging

8a. Attempting to debug a bound application.

You can't debug a bound application. The 4GWBIND utility (included with DOS/4GW Professional) will allow you to take apart a bound application so that you can debug it:

```
4GWBIND -U <boundapp.exe> <yourapp.exe>
```

8b. Debugging with an old version of the Open Watcom debugger.

DOS/4GW supports versions 8.5 and up of the Open Watcom C, C++ and FORTRAN compilers. However, in order to debug your unbound application with a Open Watcom debugger, you must have version 9.5a or later of the debugger.

If you have an older version of the debugger, we strongly recommend that you contact Open Watcom to upgrade your compiler and tools. The only way to debug a DOS/4GW Professional application with an old version of the debugger is to rename 4GWPRO.EXE to DOS4GW.EXE and make sure that it's either in the current directory or the first DOS4GW.EXE on the DOS PATH.

Tenberry will not provide technical support for this configuration; it's up to you to keep track of which DOS extender is which.

8c. Meaning of "unexpected interrupt" message/error 2001.

In version 1.95 of DOS/4GW, we revised the "unexpected interrupt" message to make it easier to understand.

For example, the message:

```
Unexpected interrupt 0E (code 0) at 168:10421034
```

is now printed:

```
error (2001): exception 0Eh (page fault) at 168:10421034
```

followed by a register dump, as before.

This message indicates that the processor detected some form of programming error and signaled an exception, which DOS/4GW trapped and reported. Exceptions which can be trapped include:

```
00h     divide by zero
01h     debug exception OR null pointer used
03h     breakpoint
04h     overflow
05h     bounds
06h     invalid opcode
07h     device not available
08h     double fault
09h     overrun
0Ah     invalid TSS
0Bh     segment not present
0Ch     stack fault
0Dh     general protection fault
0Eh     page fault
```

When you receive this message, this is the recommended course of action:

1. Record all of the information from the register dump.

2. Determine the circumstances under which your program fails.

3. Consult your debugger manual, or an Intel 386, 486 or Pentium Programmer's Reference Manual, to determine the circumstances under which the processor will generate the reported exception.

4. Get the program to fail under your debugger, which should stop the program as soon as the exception occurs.

5. Determine from the exception context why the processor generated an exception in this particular instance.

8d. Meaning of "transfer stack overflow" message/error 2002.

In version 1.95 of DOS/4GW, we added more information to the "transfer stack overflow" message. The message (which is now followed by a register dump) is printed:

```
error (2002): transfer stack overflow
on interrupt <number> at <address>
```

This message means DOS/4GW detected an overflow on its interrupt handling stack. It usually indicates either a recursive fault, or a hardware interrupt handler that can't keep up with the rate at which interrupts are occurring. The best way to understand the problem is to use the VERBOSE option in DOS/4GW to dump the interrupt history on the transfer stack; see (8e) below.

8e. Making the most of a DOS/4GW register dump.

If you can't understand your problem by running it under a debugger, the DOS/4GW register dump is your best debugging tool. To maximize the information available for postmortem debugging, set the environment variable DOS4G to VERBOSE, then reproduce the crash and record the output.

Here's a typical register dump with VERBOSE turned on, with annotations.

```
1 DOS/4GW error (2001): exception 0Eh (page fault)
                                              at 170:0042C1B2
2 TSF32: prev_ tsf32 67D8
3 SS       178 DS       178 ES       178 FS         0 GS        20
  EAX 1F000000 EBX        0 ECX   43201C EDX        E
  ESI        E EDI        0 EBP   431410 ESP    4313FC
  CS:IP  170:0042C1B2 ID 0E COD        0 FLG    10246
4 CS=  170, USE32, page granular, limit FFFFFFFF, base        0, acc CF9B
  SS=  178, USE32, page granular, limit FFFFFFFF, base        0, acc CF93
  DS=  178, USE32, page granular, limit FFFFFFFF, base        0, acc CF93
  ES=  178, USE32, page granular, limit FFFFFFFF, base        0, acc CF93
  FS=    0, USE16, byte granular, limit        0, base       15, acc  0
  GS=   20, USE16, byte granular, limit     FFFF, base     6AA0, acc 93
5 CR0: PG:1 ET:1 TS:0 EM:0 MP:0 PE:1   CR2: 1F000000 CR3: 9067
6 Crash address (unrelocated) = 1:000001B2
7 Opcode stream: 8A 18 31 D2 88 DA EB 0E 50 68 39 00 43 00 E8 1D
  Stack:
8 0178:004313FC 000E 0000 0000 0000 C2D5 0042 C057 0042 0170 0000 0000 0000
  0178:00431414 0450 0043 0452 0043 0000 0000 1430 0043 CBEF 0042 011C 0000
  0178:0043142C C568 0042 0000 0000 0000 0000 0000 0000 F248 0042 F5F8 0042
  0178:00431444 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
  0178:0043145C 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
  0178:00431474 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
9 Last 4 ints: 21 @ 170:42CF48/21 @ 170:42CF48/21 @ 170:42CF48/E @ 170:42C1B2/
```

1. The error message includes a synopsis of the problem. In this case, the processor signaled a page fault exception while executing at address 170:0042C1B2.

2. The prev_tsf32 field is not usually of interest.

3. These are the register values at the time of the exception. The interrupt number and error code (pushed on the stack by the processor for certain exceptions) are also printed.

4. The descriptors referenced by each segment register are described for your convenience. USE32 segments in general belong to your program; USE16 segments generally belong to the DOS extender. Here, CS points to your program's code segment, and SS, DS, and ES point to your data segment. FS is NULL and GS points to a DOS extender segment.

5. The control register information is not of any general interest, except on a page fault, when CR2 contains the address value that caused the fault. Since EAX in this case contains the same value, an attempt to dereference EAX could have caused this particular fault.

6. If the crash address (unrelocated) appears, it tells you where the crash occurred relative to your program's link map. You can therefore tell where a crash occurred even if you can't reproduce the crash in a debugger.

7. The opcode stream, if it appears, shows the next 16 bytes from the code segment at the point of the exception. If you disassemble these instructions, you can tell what instructions caused the crash, even without using a debugger. In this case, 8A 18 is the instruction `mov bl,[eax]`.

8. 72 words from the top of the stack, at the point of the exception, may be listed next. You may be able to recognize function calls or data from your program on the stack.

9. The four interrupts least to most recently handled by DOS/4GW in protected mode are listed next. In this example, the last interrupt issued before the page fault occurred was an INT 21h (DOS call) at address 170:42CF48. Sometimes, this information provides helpful context.

Here's an abridged register dump from a stack overflow.

```
  DOS/4GW error (2002): transfer stack overflow
                                on interrupt 70h at 170:0042C002
  TSF32: prev_ tsf32 48C8
  SS        C8 DS       170 ES        28 FS         0 GS        20
  EAX AAAAAAAA EBX BBBBBBBB ECX CCCCCCCC EDX DDDDDDDD
  ESI 51515151 EDI D1D1D1D1 EBP B1B1B1B1 ESP     4884
1 CS:IP  170:0042C002 ID 70 COD        0 FLG        2
  ...
2 Previous TSF:
  TSF32: prev_ tsf32 498C
  SS        C8 DS       170 ES        28 FS         0 GS        20
  EAX AAAAAAAA EBX BBBBBBBB ECX CCCCCCCC EDX DDDDDDDD
  ESI 51515151 EDI D1D1D1D1 EBP B1B1B1B1 ESP     4960
3 CS:IP  170:0042C002 ID 70 COD        0 FLG        2
  ...
  Previous TSF:
  TSF32: prev_ tsf32 67E4
  SS       178 DS       170 ES        28 FS         0 GS        20
  EAX AAAAAAAA EBX BBBBBBBB ECX CCCCCCCC EDX DDDDDDDD
  ESI 51515151 EDI D1D1D1D1 EBP B1B1B1B1 ESP   42FFE0
4 CS:IP  170:0042C039 ID 70 COD        0 FLG      202
5 Opcode stream: CF 66 B8 62 25 66 8C CB 66 8E DB BA 00 C0 42 00
  Last 4 ints: 70 @ 170:42C002/70 @ 170:42C002/70 @ 170:42C002/70 @ 170:42C002/
```

1. We overflowed the transfer stack while trying to process an interrupt 70h at 170:0042C002.

2. The entire interrupt history from the transfer stack is printed next. The prev_tsf32 numbers increase as we progress from most recent to least recent interrupt. All of these interrupts are still pending, which is why we ran out of stack space.

3. Before we overflowed the stack, we got the same interrupt at the same address. For a recursive interrupt situation, this is typical.

4. The oldest frame on the transfer stack shows the recursion was touched off at a slightly different address. Looking at this address may help you understand the recursion.

5. The opcode stream and last four interrupt information comes from the newest transfer stack frame, not the oldest.

14.9 Compatibility

9a. Running DOS/4GW applications from inside Lotus 1-2-3.

In order to run DOS/4GW applications while "shelled out" from Lotus 1-2-3, you must use the PRIVATXM program included with your Open Watcom compiler. Otherwise, 1-2-3 will take all of the memory on your machine and prevent DOS/4GW from using it.

Before starting 1-2-3, you must set the DOS16M environment variable to limit Lotus' memory use (see your Open Watcom manual). After shelling out, you must run PRIVATXM, then clear the DOS16M environment variable before running your application.

9b. EMM386.EXE provided with DOS 6.0.

We know of at least three serious bugs in the EMM386.EXE distributed with MS-DOS 6.0, one involving mis-counting the amount of available memory, one involving mapping too little of the High Memory Area (HMA) into its page tables, and one involving allocation of EMS memory. Version 1.95 of DOS/4GW contains workarounds for some of these problems.

If you are having problems with DOS/4GW and you are using an EMM386.EXE dated 3-10-93 at 6:00:00, or later, you may wish to try the following workarounds, in sequence, until the problem goes away.

- Configure EMM386 with both the NOEMS and NOVCPI options.
- Convert the DEVICEHIGH statements in your CONFIG.SYS to DEVICE statements, and remove the LH (Load High) commands from your AUTOEXEC.BAT.
- Run in a Windows DOS box.
- Replace EMM386 with another memory manager, such as QEMM-386, 386Max, or an older version of EMM386.
- Run with HIMEM.SYS alone.

You may also wish to contact Microsoft Corporation to inquire about the availability of a fix.

9c. Spawning under OS/2 2.1.

We know of a bug in OS/2 2.1 that prevents one DOS/4GW application from spawning another over and over again. The actual number of repeated spawns that are possible under OS/2 varies from machine to machine, but is generally about 30.

This bug also affects programs running under other DOS extenders, and we have not yet found a workaround, other than linking your two programs together as a single program.

9d. "DPMI host error: cannot lock stack".

This error message almost always indicates insufficient memory, rather than a real incompatibility. If you see it under an OS/2 DOS box, you probably need to edit your DOS Session settings and make DPMI_MEMORY_LIMIT larger.

9e. Bug in Novell TCPIP driver.

Some versions of a program from Novell called TCPIP.EXE, a real-mode program, will cause the high words of EAX and EDX to be altered during a hardware interrupt. This bug breaks protected-mode software (and other real-mode software that uses the 80386 registers). Novell has released a newer version of TCPIP that fixes the problem; contact Novell to obtain the fix.

9f. Bugs in Windows NT.

The initial release of Windows NT includes a DPMI host, DOSX.EXE, with several serious bugs, some of which apparently cannot be worked around. We cannot warranty operation of DOS/4GW under Windows NT at this time, but we are continuing to exercise our best efforts to work around these problems.

You may wish to contact Microsoft Corporation to inquire about the availability of a new version of DOSX.EXE.

Windows 3.x Programming Guide

15 Creating 16-bit Windows 3.x Applications

This chapter describes how to compile and link 16-bit Windows 3.x applications simply and quickly. In this chapter, we look at applications written to exploit the Windows 3.x Application Programming Interface (API).

We will illustrate the steps to creating 16-bit Windows 3.x applications by taking a small sample application and showing you how to compile, link, run and debug it.

Note - It is supposed you are working on the host with Windows 3.x installed. If you are on the host with any other operating system you should setup INCLUDE environment variable correctly to compile for 16-bit Windows 3.x target.

You can do that by command (DOS, OS/2, NT)

set INCLUDE=%WATCOM%\h;%WATCOM%\h\win

or by command (LINUX)

export INCLUDE=$WATCOM/h:$WATCOM/h/win

15.1 The Sample GUI Application

To demonstrate the creation of 16-bit Windows 3.x applications, we introduce a simple sample program. The following example is the "hello" program adapted for Windows.

```
#include <windows.h>

int PASCAL WinMain( HANDLE hInstance, HANDLE hPrevInst,
                    LPSTR lpCmdLine, int nCmdShow )
{
    MessageBox( NULL, "Hello world",
                "Open Watcom C/C++ for Windows",
                MB_ OK | MB_ TASKMODAL );
    return( 0 );
}
```

The goal of this program is to display the message "Hello world" on the screen. The `MessageBox` Windows API function is used to accomplish this task. We will take you through the steps necessary to produce this result.

15.2 Building and Running the GUI Application

To compile and link our example program which is stored in the file `hello.c`, enter the following command:

```
C>wcl -l=windows -bt=windows hello.c
```

The typical messages that appear on the screen are shown in the following illustration.

```
C>wcl -l=windows -bt=windows hello.c
Open Watcom C/C++16 Compile and Link Utility
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1988-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
        wcc hello.c  -bt=windows
Open Watcom C16 Optimizing Compiler
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1984-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
hello.c: 10 lines, included 6500, 0 warnings, 0 errors
Code size: 37

Open Watcom Linker
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1985-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
loading object files
searching libraries
creating a Windows 16-bit executable
```

If you examine the current directory, you will find that two files have been created. These are `hello.obj` (the result of compiling `hello.c`) and `hello.exe` (the result of linking `hello.obj` with the appropriate Open Watcom C/C++ libraries).

The resultant 16-bit Windows 3.x application `HELLO.EXE` can now be run under Windows 3.x.

15.3 Debugging the GUI Application

Let us assume that you wish to debug your application in order to locate an error in programming. In the previous section, the "hello" program was compiled with default compile and link options. When debugging an application, it is useful to refer to the symbolic names of routines and variables. It is also convenient to debug at the source line level rather than the machine language level. To do this, we must direct both the compiler and linker to include additional debugging information in the object and executable files. Using the ***WCL*** command, this is fairly straightforward. ***WCL*** recognizes the Open Watcom C/C++ compiler "debug" options and will create the appropriate debug directives for the Open Watcom Linker.

For example, to compile and link the "hello" program with debugging information, the following command may be issued.

```
C>wcl -l=windows -bt=windows -d2 hello.c
```

The typical messages that appear on the screen are shown in the following illustration.

```
C>wcl -l=windows -bt=windows -d2 hello.c
Open Watcom C/C++16 Compile and Link Utility
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1988-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
        wcc hello.c -bt=windows -d2
Open Watcom C16 Optimizing Compiler
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1984-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
hello.c: 10 lines, included 6500, 0 warnings, 0 errors
Code size: 58

Open Watcom Linker
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1985-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
loading object files
searching libraries
creating a Windows 16-bit executable
```

The "d2" option requests the maximum amount of debugging information that can be provided by the Open Watcom C/C++ compiler. ***WCL*** will make sure that this debugging information is included in the executable file that is produced by the linker.

The "Code size" value is larger than in the previous example since selection of the "d2" option results in fewer code optimizations by default. You can request more optimization by specifying the appropriate options. However, you do so at the risk of making it more difficult for yourself to determine the relationship between the object code and the original source language code.

To request the Open Watcom Debugger to assist in debugging the application, select the Open Watcom Debugger icon. It would be too ambitious to describe the debugger in this introductory chapter so we refer you to the book entitled ***Open Watcom Debugger User's Guide.***

16 Porting Non-GUI Applications to 16-bit Windows 3.x

Generally, an application that is to run in a windowed environment must be written in such a way as to exploit the Windows Application Programming Interface (API). To take an existing character-based (i.e., non-graphical) application that ran under a system such as DOS and adapt it to run under Windows can require some considerable effort. There is a steep learning curve associated with the API function libraries.

This chapter describes how to create a Windows application quickly and simply from an application that does not use the Windows API. The application will make use of Open Watcom's default windowing support.

Suppose you have a set of C/C++ applications that previously ran under a system like DOS and you now wish to run them under Windows 3.x. To achieve this, you can simply recompile your application with the appropriate options and link with the appropriate libraries. We provide a default windowing system that turns your character-mode application into a simple Windows 3.x Graphical User Interface (GUI) application.

Normally, a Windows 3.x GUI application makes use of user-interface tools such as menus, icons, scroll bars, etc. However, an application that was not designed as a windowed application (such as a DOS application) can run as a GUI application. This is achieved by our default windowing system. The following sections describe the default windowing system.

16.1 Console Device in a Windowed Environment

In a C/C++ application that runs under DOS, *stdin (C++ cin)* and *stdout (C++ cout)* are connected to the standard input and standard output devices respectively. It is not a recommended practice to read directly from the standard input device or write to the standard output device when running in a windowed environment. For this reason, a default windowing environment is created for C/C++ applications that read from *stdin (C++ cin)* or write to *stdout (C++ cout).* When your application is started, a window is created in which output to *stdout (C++ cout)* is displayed and input from *stdin (C++ cin)* is requested.

In addition to the standard I/O device, it is also possible to perform I/O to the console by explicitly opening a file whose name is "CON". When this occurs, another window is created and displayed. This window is different from the one created for standard input and standard output. In fact, every time you open the console device a different window is created. This provides a simple multi-windowing system for multiple streams of data to and from the console device.

16.2 The Sample Non-GUI Application

To demonstrate the creation of 16-bit Windows 3.x applications, we introduce a simple sample program. For our example, we are going to use the famous "hello" program.

```
#include <stdio.h>

void main()
{
    printf( "Hello world\n" );
}
```

The C++ version of this program follows:

```
#include <iostream.h>

void main()
{
    cout << "Hello world" << endl;
}
```

The goal of this program is to display the message "Hello world" on the screen. The C version uses the C library `printf` routine to accomplish this task. The C++ version uses the "iostream" library to accomplish this task. We will take you through the steps necessary to produce this result.

16.3 Building and Running the Non-GUI Application

Very little effort is required to port an existing C/C++ application to Windows 3.x.

You must compile and link the file `hello.c` specifying the "bw" option.

```
C>wcl -l=windows -bw -bt=windows hello.c
```

The typical messages that appear on the screen are shown in the following illustration.

```
C>wcl -l=windows -bw -bt=windows hello.c
Open Watcom C/C++16 Compile and Link Utility
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1988-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
        wcc hello.c  -bw -bt=windows
Open Watcom C16 Optimizing Compiler
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1984-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
hello.c: 6 lines, included 155, 0 warnings, 0 errors
Code size: 17

Open Watcom Linker
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1985-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
loading object files
searching libraries
creating a Windows 16-bit executable
```

If you examine the current directory, you will find that two files have been created. These are `hello.obj` (the result of compiling `hello.c`) and `hello.exe` (the result of linking `hello.obj` with the appropriate Open Watcom C/C++ libraries).

The resultant 16-bit Windows 3.x application `HELLO.EXE` can now be run under Windows 3.x as a Windows GUI application.

16.4 Debugging the Non-GUI Application

Let us assume that you wish to debug your application in order to locate an error in programming. In the previous section, the "hello" program was compiled with default compile and link options. When debugging an application, it is useful to refer to the symbolic names of routines and variables. It is also convenient to debug at the source line level rather than the machine language level. To do this, we must direct both the compiler and linker to include additional debugging information in the object and executable files. Using the ***WCL*** command, this is fairly straightforward. ***WCL*** recognizes the Open Watcom C/C++ compiler "debug" options and will create the appropriate debug directives for the Open Watcom Linker.

For example, to compile and link the "hello" program with debugging information, the following command may be issued.

```
C>wcl -l=windows -bw -bt=windows -d2 hello.c
```

The typical messages that appear on the screen are shown in the following illustration.

```
C>wcl -l=windows -bw -bt=windows -d2 hello.c
Open Watcom C/C++16 Compile and Link Utility
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1988-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
        wcc hello.c -bw -bt=windows -d2
Open Watcom C16 Optimizing Compiler
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1984-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
hello.c: 6 lines, included 155, 0 warnings, 0 errors
Code size: 23

Open Watcom Linker
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1985-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
loading object files
searching libraries
creating a Windows 16-bit executable
```

The "d2" option requests the maximum amount of debugging information that can be provided by the Open Watcom C/C++ compiler. ***WCL*** will make sure that this debugging information is included in the executable file that is produced by the linker.

The "Code size" value is larger than in the previous example since selection of the "d2" option results in fewer code optimizations by default. You can request more optimization by specifying the appropriate options. However, you do so at the risk of making it more difficult for yourself to determine the relationship between the object code and the original source language code.

To request the Open Watcom Debugger to assist in debugging the application, select the Open Watcom Debugger icon. It would be too ambitious to describe the debugger in this introductory chapter so we refer you to the book entitled ***Open Watcom Debugger User's Guide.***

16.5 Default Windowing Library Functions

A few library functions have been written to enable some simple customization of the default windowing system's behaviour. The following functions are supplied:

_dwDeleteOnClose

```
int _dwDeleteOnClose( int handle );
```

This function tells the console window that it should close itself when the file is closed. You must pass to it the handle associated with the opened console.

_dwSetAboutDlg

```
int _dwSetAboutDlg( const char *title, const char *text );
```

This function sets the about dialog box of the default windowing system. The "title" points to the string that will replace the current title. If title is NULL then the title will not be replaced. The "text" points to a string which will be placed in the about box. To get multiple lines, embed a new line after each logical line in the string. If "text" is NULL, then the current text in the about box will not be replaced.

_dwSetAppTitle

```
int _dwSetAppTitle( const char *title );
```

This function sets the main window's title.

_dwSetConTitle

```
int _dwSetConTitle( int handle, const char *title );
```

This function sets the console window's title which corresponds to the handle passed to it.

_dwShutDown

```
int _dwShutDown( void );
```

This function shuts down the default windowing I/O system. The application will continue to execute but no windows will be available for output.

_dwYield

```
int _dwYield( void );
```

This function yields control back to the operating system, thereby giving other processes a chance to run.

These functions are described more fully in the ***WATCOM C Library Reference.***

17 Creating 32-bit Windows 3.x Applications

This chapter describes how to compile and link 32-bit Windows 3.x applications simply and quickly. In this chapter, we look at applications written to exploit the Windows 3.x Application Programming Interface (API).

We will illustrate the steps to creating 32-bit Windows 3.x applications by taking a small sample application and showing you how to compile, link, run and debug it.

Note - It is supposed you are working on the host with Windows 3.x installed. If you are on the host with any other operating system you should setup INCLUDE environment variable correctly to compile for 32-bit Windows 3.x target.

You can do that by command (DOS, OS/2, NT)

set INCLUDE=%WATCOM%\h;%WATCOM%\h\win

or by command (LINUX)

export INCLUDE=$WATCOM/h:$WATCOM/h/win

17.1 The Sample GUI Application

To demonstrate the creation of 32-bit Windows 3.x applications, we introduce a simple sample program. The following example is the "hello" program adapted for Windows.

```
#include <windows.h>

int PASCAL WinMain( HANDLE hInstance, HANDLE hPrevInst,
                    LPSTR lpCmdLine, int nCmdShow )
{
    MessageBox( NULL, "Hello world",
                "Open Watcom C/C++ for Windows",
                MB_ OK | MB_ TASKMODAL );
    return( 0 );
}
```

The goal of this program is to display the message "Hello world" on the screen. The `MessageBox` Windows API function is used to accomplish this task. We will take you through the steps necessary to produce this result.

17.2 Building and Running the GUI Application

To compile and link our example program which is stored in the file `hello.c`, enter the following command:

```
C>wcl386 -l=win386 -bt=windows hello.c
```

The typical messages that appear on the screen are shown in the following illustration.

```
C>wcl386 -l=win386 -bt=windows hello.c
Open Watcom C/C++32 Compile and Link Utility
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1988-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
        wcc386 hello.c  -bt=windows
Open Watcom C32 Optimizing Compiler
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1984-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
hello.c: 10 lines, included 6500, 0 warnings, 0 errors
Code size: 41

Open Watcom Linker
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1985-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
loading object files
searching libraries
creating a Windows 32-bit executable
```

If you examine the current directory, you will find that two files have been created. These are `hello.obj` (the result of compiling `hello.c`) and `hello.rex` (the result of linking `hello.obj` with the appropriate Open Watcom C/C++ libraries). The ".rex" file must now be combined with Open Watcom's 32-bit Windows supervisor `WIN386.EXT` using the Open Watcom Bind utility. `WBIND.EXE` combines your 32-bit application code and data (".rex" file) with the 32-bit Windows supervisor. The process involves the following steps:

1. `WBIND` copies `WIN386.EXT` into the current directory.

2. `WBIND.EXE` optionally runs the resource compiler on the 32-bit Windows supervisor so that the 32-bit executable can have access to the applications resources.

3. `WBIND.EXE` concatenates `WIN386.EXT` and the ".rex" file, and creates a ".exe" file with the same name as the ".rex" file.

The following describes the syntax of the `WBIND` command.

WBIND file_spec [-d] [-n] [-q] [-s supervisor] [-R rc_options]

The square brackets [ ] denote items which are optional.

WBIND	is the name of the Open Watcom Bind utility.
file_spec	is the name of the 32-bit Windows EXE to bind.
-d	requests that a 32-bit DLL be built.
-n	indicates that the resource compiler is NOT to be invoked.
-q	requests that WBIND run in quiet mode (no informational messages are displayed).
-s supervisor	specifies the path and name of the Windows supervisor to be bound with the application. If not specified, a search of the paths listed in the **PATH** environment variable is performed. If this search is not successful and the **WATCOM** environment variable is defined, the `%WATCOM%\BINW` directory is searched.
-R rc_options	all options after -R are passed to the resource compiler.

To bind our example program, the following command may be issued:

```
C>wbind hello -n
```

If the "s" option is specified, it must identify the location of the `WIN386.EXT` file or the `W386DLL.EXT` file (if you are building a DLL).

Example:

```
C>wbind hello -n -s c:\watcom\binw\win386.ext
```

If the "s" option is not specified, then the **WATCOM** environment variable must be defined or the "BINW" directory must be listed in your **PATH** environment variable.

Example:

```
C>set watcom=c:\watcom
  or
C>path c:\watcom\binw;c:\dos;c:\windows
```

The resultant 32-bit Windows 3.x application `HELLO.EXE` can now be run under Windows 3.x.

17.3 Debugging the GUI Application

Let us assume that you wish to debug your application in order to locate an error in programming. In the previous section, the "hello" program was compiled with default compile and link options. When debugging an application, it is useful to refer to the symbolic names of routines and variables. It is also convenient to debug at the source line level rather than the machine language level. To do this, we must direct both the compiler and linker to include additional debugging information in the object and executable files. Using the ***WCL386*** command, this is fairly straightforward. ***WCL386*** recognizes the Open Watcom C/C++ compiler "debug" options and will create the appropriate debug directives for the Open Watcom Linker.

For example, to compile and link the "hello" program with debugging information, the following command may be issued.

```
C>wcl386 -l=win386 -bt=windows -d2 hello.c
```

The typical messages that appear on the screen are shown in the following illustration.

```
C>wcl386 -l=win386 -bt=windows -d2 hello.c
Open Watcom C/C++32 Compile and Link Utility
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1988-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
        wcc386 hello.c -bt=windows -d2
Open Watcom C32 Optimizing Compiler
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1984-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
hello.c: 10 lines, included 6500, 0 warnings, 0 errors
Code size: 66

Open Watcom Linker
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1985-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
loading object files
searching libraries
creating a Windows 32-bit executable
```

The "d2" option requests the maximum amount of debugging information that can be provided by the Open Watcom C/C++ compiler. ***WCL386*** will make sure that this debugging information is included in the executable file that is produced by the linker.

The "Code size" value is larger than in the previous example since selection of the "d2" option results in fewer code optimizations by default. You can request more optimization by specifying the appropriate options. However, you do so at the risk of making it more difficult for yourself to determine the relationship between the object code and the original source language code.

Once again, the ".rex" file must be combined with Open Watcom's 32-bit Windows supervisor `WIN386.EXT` using the Open Watcom Bind utility. This step is described in the previous section.

To request the Open Watcom Debugger to assist in debugging the application, select the Open Watcom Debugger icon. It would be too ambitious to describe the debugger in this introductory chapter so we refer you to the book entitled ***Open Watcom Debugger User's Guide.***

18 Porting Non-GUI Applications to 32-bit Windows 3.x

Generally, an application that is to run in a windowed environment must be written in such a way as to exploit the Windows Application Programming Interface (API). To take an existing character-based (i.e., non-graphical) application that ran under a system such as DOS and adapt it to run under Windows can require some considerable effort. There is a steep learning curve associated with the API function libraries.

This chapter describes how to create a Windows application quickly and simply from an application that does not use the Windows API. The application will make use of Open Watcom's default windowing support.

Suppose you have a set of C/C++ applications that previously ran under a system like DOS and you now wish to run them under Windows 3.x. To achieve this, you can simply recompile your application with the appropriate options and link with the appropriate libraries. We provide a default windowing system that turns your character-mode application into a simple Windows 3.x Graphical User Interface (GUI) application.

Normally, a Windows 3.x GUI application makes use of user-interface tools such as menus, icons, scroll bars, etc. However, an application that was not designed as a windowed application (such as a DOS application) can run as a GUI application. This is achieved by our default windowing system. The following sections describe the default windowing system.

18.1 Console Device in a Windowed Environment

In a C/C++ application that runs under DOS, *stdin (C++ cin)* and *stdout (C++ cout)* are connected to the standard input and standard output devices respectively. It is not a recommended practice to read directly from the standard input device or write to the standard output device when running in a windowed environment. For this reason, a default windowing environment is created for C/C++ applications that read from *stdin (C++ cin)* or write to *stdout (C++ cout).* When your application is started, a window is created in which output to *stdout (C++ cout)* is displayed and input from *stdin (C++ cin)* is requested.

In addition to the standard I/O device, it is also possible to perform I/O to the console by explicitly opening a file whose name is "CON". When this occurs, another window is created and displayed. This window is different from the one created for standard input and standard output. In fact, every time you open the console device a different window is created. This provides a simple multi-windowing system for multiple streams of data to and from the console device.

18.2 The Sample Non-GUI Application

To demonstrate the creation of 32-bit Windows 3.x applications, we introduce a simple sample program. For our example, we are going to use the famous "hello" program.

```
#include <stdio.h>

void main()
{
    printf( "Hello world\n" );
}
```

The C++ version of this program follows:

```
#include <iostream.h>

void main()
{
    cout << "Hello world" << endl;
}
```

The goal of this program is to display the message "Hello world" on the screen. The C version uses the C library `printf` routine to accomplish this task. The C++ version uses the "iostream" library to accomplish this task. We will take you through the steps necessary to produce this result.

18.3 Building and Running the Non-GUI Application

Very little effort is required to port an existing C/C++ application to Windows 3.x.

You must compile and link the file `hello.c` specifying the "bw" option.

```
C>wcl386 -l=win386 -bw -bt=windows hello.c
```

The typical messages that appear on the screen are shown in the following illustration.

```
C>wcl386 -l=win386 -bw -bt=windows hello.c
Open Watcom C/C++32 Compile and Link Utility
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1988-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
        wcc386 hello.c  -bw -bt=windows
Open Watcom C32 Optimizing Compiler
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1984-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
hello.c: 6 lines, included 174, 0 warnings, 0 errors
Code size: 24

Open Watcom Linker
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1985-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
loading object files
searching libraries
creating a Windows 32-bit executable
```

If you examine the current directory, you will find that two files have been created. These are `hello.obj` (the result of compiling `hello.c`) and `hello.rex` (the result of linking `hello.obj` with the appropriate Open Watcom C/C++ libraries). The ".rex" file must now be combined with Open Watcom's 32-bit Windows supervisor `WIN386.EXT` using the Open Watcom Bind utility. `WBIND.EXE` combines your 32-bit application code and data (".rex" file) with the 32-bit Windows supervisor. The process involves the following steps:

1. `WBIND` copies `WIN386.EXT` into the current directory.
2. `WBIND.EXE` optionally runs the resource compiler on the 32-bit Windows supervisor so that the 32-bit executable can have access to the applications resources.
3. `WBIND.EXE` concatenates `WIN386.EXT` and the ".rex" file, and creates a ".exe" file with the same name as the ".rex" file.

The following describes the syntax of the `WBIND` command.

WBIND file_spec [-d] [-n] [-q] [-s supervisor] [-R rc_options]

The square brackets [ ] denote items which are optional.

WBIND	is the name of the Open Watcom Bind utility.
file_spec	is the name of the 32-bit Windows EXE to bind.
-d	requests that a 32-bit DLL be built.
-n	indicates that the resource compiler is NOT to be invoked.
-q	requests that WBIND run in quiet mode (no informational messages are displayed).
-s supervisor	specifies the path and name of the Windows supervisor to be bound with the application. If not specified, a search of the paths listed in the **PATH** environment variable is performed. If this search is not successful and the **WATCOM** environment variable is defined, the `%WATCOM%\BINW` directory is searched.
-R rc_options	all options after -R are passed to the resource compiler.

To bind our example program, the following command may be issued:

```
C>wbind hello -n
```

If the "s" option is specified, it must identify the location of the `WIN386.EXT` file or the `W386DLL.EXT` file (if you are building a DLL).

Example:

```
C>wbind hello -n -s c:\watcom\binw\win386.ext
```

If the "s" option is not specified, then the **WATCOM** environment variable must be defined or the "BINW" directory must be listed in your **PATH** environment variable.

Example:

```
C>set watcom=c:\watcom
  or
C>path c:\watcom\binw;c:\dos;c:\windows
```

The resultant 32-bit Windows 3.x application `HELLO.EXE` can now be run under Windows 3.x as a Windows GUI application.

18.4 Debugging the Non-GUI Application

Let us assume that you wish to debug your application in order to locate an error in programming. In the previous section, the "hello" program was compiled with default compile and link options. When debugging an application, it is useful to refer to the symbolic names of routines and variables. It is also convenient to debug at the source line level rather than the machine language level. To do this, we must direct both the compiler and linker to include additional debugging information in the object and executable files. Using the ***WCL386*** command, this is fairly straightforward. ***WCL386*** recognizes the Open Watcom C/C++ compiler "debug" options and will create the appropriate debug directives for the Open Watcom Linker.

For example, to compile and link the "hello" program with debugging information, the following command may be issued.

```
C>wcl386 -l=win386 -bw -bt=windows -d2 hello.c
```

The typical messages that appear on the screen are shown in the following illustration.

```
C>wcl386 -l=win386 -bw -bt=windows -d2 hello.c
Open Watcom C/C++32 Compile and Link Utility
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1988-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
        wcc386 hello.c -bw -bt=windows -d2
Open Watcom C32 Optimizing Compiler
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1984-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
hello.c: 6 lines, included 174, 0 warnings, 0 errors
Code size: 45

Open Watcom Linker
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1985-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
loading object files
searching libraries
creating a Windows 32-bit executable
```

The "d2" option requests the maximum amount of debugging information that can be provided by the Open Watcom C/C++ compiler. ***WCL386*** will make sure that this debugging information is included in the executable file that is produced by the linker.

The "Code size" value is larger than in the previous example since selection of the "d2" option results in fewer code optimizations by default. You can request more optimization by specifying the appropriate options. However, you do so at the risk of making it more difficult for yourself to determine the relationship between the object code and the original source language code.

Once again, the ".rex" file must be combined with Open Watcom's 32-bit Windows supervisor `WIN386.EXT` using the Open Watcom Bind utility. This step is described in the previous section.

To request the Open Watcom Debugger to assist in debugging the application, select the Open Watcom Debugger icon. It would be too ambitious to describe the debugger in this introductory chapter so we refer you to the book entitled ***Open Watcom Debugger User's Guide.***

18.5 Default Windowing Library Functions

A few library functions have been written to enable some simple customization of the default windowing system's behaviour. The following functions are supplied:

_dwDeleteOnClose

```
int _dwDeleteOnClose( int handle );
```

This function tells the console window that it should close itself when the file is closed. You must pass to it the handle associated with the opened console.

_dwSetAboutDlg

```
int _dwSetAboutDlg( const char *title, const char *text );
```

This function sets the about dialog box of the default windowing system. The "title" points to the string that will replace the current title. If title is NULL then the title will not be replaced. The "text" points to a string which will be placed in the about box. To get multiple lines, embed a new line after each logical line in the string. If "text" is NULL, then the current text in the about box will not be replaced.

_dwSetAppTitle

```
int _dwSetAppTitle( const char *title );
```

This function sets the main window's title.

_dwSetConTitle

```
int _dwSetConTitle( int handle, const char *title );
```

This function sets the console window's title which corresponds to the handle passed to it.

_dwShutDown

```
int _dwShutDown( void );
```

This function shuts down the default windowing I/O system. The application will continue to execute but no windows will be available for output.

_dwYield

```
int _ dwYield( void );
```

This function yields control back to the operating system, thereby giving other processes a chance to run.

These functions are described more fully in the ***WATCOM C Library Reference.***

19 The Open Watcom 32-bit Windows 3.x Extender

Open Watcom C/C++ contains the necessary tools and libraries to create 32-bit applications for Windows 3.x. Using Open Watcom C/C++ gives the programmer the benefits of a 32-bit flat memory model and access to the full Windows API (along with the usual C/C++ library functions).

The general model of the environment is as follows: The 32-bit flat memory model program is linked against a special 32-bit Windows library. This library contains the necessary information to invoke special 16-bit functions, which lie in the supervisor `(WIN386.EXT)`. The 32-bit program is then bound (using `WBIND.EXE`) with the supervisor to create a Windows executable. At the same time as the 32-bit program is being bound, the resource compiler is run on the supervisor, and all the resources for the application are placed there. When the application is started, the supervisor obtains the 32-bit memory, relocates the 32-bit application into the memory, and invokes the 32-bit application.

All Windows functions are invoked from the supervisor, and all callback routines lie within the supervisor. The local heap resides within the supervisor as well.

If you are starting from a 16-bit Windows application, most of the code will not change when you port it to the 32-bit Windows environment. However, because of the nature of the Windows API and its implicit dependencies on a 16-bit environment, some source changes are necessary. These source changes are minimal, and are backwards compatible with the 16-bit environment.

19.1 Pointers

Throughout this document, there will be references to both ***near*** and ***far***, and ***16-bit*** and ***32-bit*** pointers. Since this can rapidly become confusing, some initial explanations will be given here.

A ***far pointer*** is a pointer that is composed of both a selector and an offset. A selector determines a specific region of memory, and the offset is relative to the start of this region. A ***near pointer*** is a pointer that has an offset only, the selector is automatically assumed by the CPU.

The problem with far pointers is the selector overhead. Using a far pointer is much more expensive than using a near pointer. This is the advantage of the 32-bit flat memory model - all pointers within the program are near, and yet you can address up to 4 gigabytes of memory.

A ***16-bit near pointer*** occupies 2 bytes of memory (i.e., the offset is 16 bits long). This pointer can reference up to 64K of data.

A ***16-bit far pointer*** occupies 4 bytes of memory. There is a 16-bit selector and a 16-bit offset. This pointer can reference up to 64K of data.

A ***32-bit near pointer*** occupies 4 bytes of memory (i.e., the offset is 32 bits long). This pointer can reference up to 4 gigabytes of data.

A ***32-bit far pointer*** occupies 6 bytes of memory. There is a 16-bit selector and a 32-bit offset. This pointer can reference up to 4 gigabytes of data.

Windows, in general, uses 16-bit far pointers to pass information around. These 16-bit far pointers can also be used by a 32-bit Windows application. Using a special macro, ***MK_FP32***, the offset of the 16-bit far pointer is extended from 16 bits to 32 bits, and the pointer becomes a 32-bit far pointer. The 32-bit far pointer is then used by the application to access the data (note that offsets still must be less than 64K, since the selector is still for a 64K data area).

19.2 Implementation Overview

This section provides an overview of the issues that require consideration when creating a 32-bit Windows application for a 16-bit Windows environment.

First, all modules have to be recompiled for the 32-bit flat memory model with a compiler capable of generating 32-bit instructions. Many Windows API functions take ***int*** as a parameter. This ***int*** is from the 16-bit world, and is 2 bytes long. In the 32-bit world, this ***int*** becomes 4 bytes long. Since Windows is only expecting two bytes of data, all occurrences of ***int*** have to be changed to ***short*** in `WINDOWS.H`.

Pointers to data passed to Windows are all far pointers. We will be passing pointers to data in our 32-bit flat address space, and these pointers are near pointers. By simply getting rid of all ***far*** keywords in `WINDOWS.H`, all pointers will now be passed as 32-bit near pointers. As well, notice that these 32-bit near pointers are the same size as as their 16-bit far pointer counterparts (4 bytes). This is good, since all data structures containing pointers will remain the same size.

Windows cannot be called from 32-bit code on a 32-bit stack. This means that in order to call the API functions, it is necessary to write a set of cover functions that will accept the parameters, switch into a 16-bit environment, and then call Windows. There is another issue, though. Windows only understands 16-bit pointers, so before calling Windows, all pointers being passed to Windows must be converted to 16-bit far pointers.

It turns out that Windows can also call back to your application. Windows can only call 16-bit code, though, so there is a need for a bridge from the 16-bit side to the 32-bit side. It is necessary to allocate 16-bit call back routines that can be passed to Windows. These call back routines will then switch into the 32-bit environment and call whatever 32-bit function is required. The 32-bit call back has to be declared as a far function, since it is necessary to issue a far call to enter it from the 16-bit side. If it is a far function, then the compiler will generate the appropriate code for it to return from the far call.

Once Windows calls you back, it can hand you 16-bit far pointers in a long (4 byte) parameter. This pointer can only be used in the 32-bit environment if it is a 32-bit far pointer, not a 16-bit far pointer. The conversion is simple: the 16-bit offset is extended to a 32-bit offset (the high word is zeroed out). Any far pointer that Windows hands to you must be converted in this way.

Sometimes, a Windows application wants to call a procedure in a DLL. The procedure address is a 16-bit far pointer. It is not possible to issue an indirect call to this address from the 32-bit environment, so some sort of interface is needed. This interface would switch into the 16-bit environment, and then call the 16-bit function.

These issues, along with other minor items, are handled by Open Watcom C/C++, and are discussed in more technical detail in later sections.

19.3 System Structure

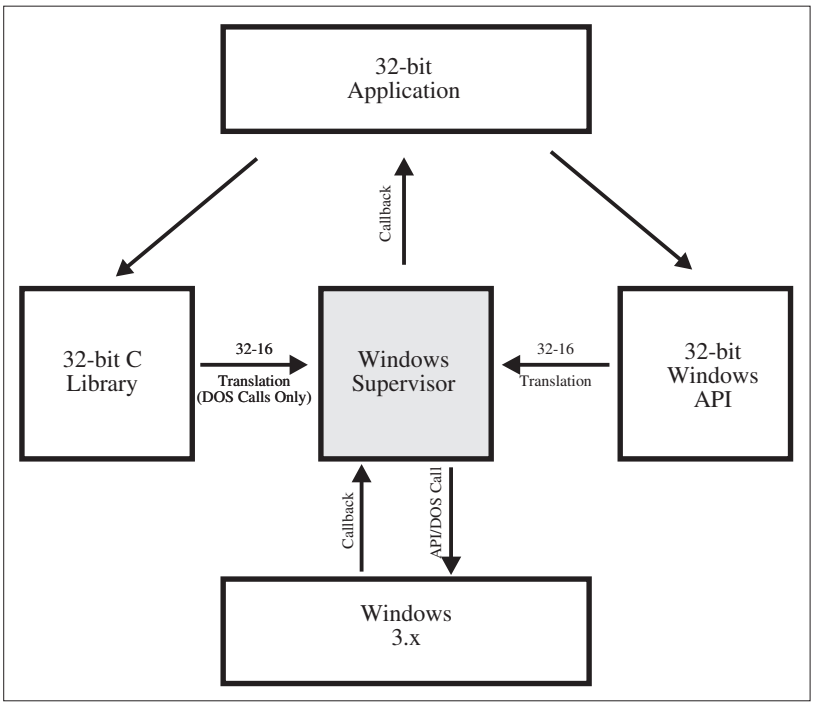


Figure 5. *WIN386 Structure*

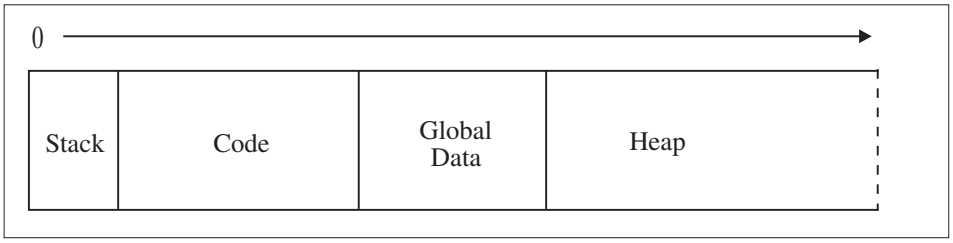


Figure 6. *32-bit Application Structure*

19.4 System Overview

- `WIN386.EXT` is the key component of a 32-bit Windows application. It is a 16-bit Windows application which contains:
    - All application resources.
    - A 16-bit local heap.
    - A 16-bit stack.
- `W386DLL.EXT` is similar to `WIN386.EXT`, only it provides a DLL interface.

`WIN386.EXT` is bound to your 32-bit application to create a 32-bit application that will run under Windows 3.x. `WIN386.EXT` provides the following functionality:

- supervisor to bring the 32-bit application into memory and start it running.
- "glue" functions to connect to Windows for both API and DOS functionality. This interface is designed to transparently set up the calling functions' pointers and parameters to their 16-bit counterparts.
- "glue-back" functions to allow Windows to call back 32-bit routines.
- special code to allow debugging of 32-bit applications.

- `WINDOWS.H` has been specially modified for use in the 32-bit Windows environment. As well, it contains all special definitions for 32-bit applications.
- `WIN386.LIB` contains all the necessary library functions to connect to the 32-bit supervisor `WIN386.EXT`. All Windows API calls and Open Watcom C/C++ library DOS calls are found here.
- The standard C/C++ library functions, specially modified to run in the 32-bit environment, are located in the `\WATCOM\LIB386\WIN` directory.
- `WBIND.EXE` merges your 32-bit executable and the appropriate Supervisor into a single executable.

19.5 Steps to Obtaining a 32-bit Application

The following is an overview of the procedure for creating a 32-bit Windows Application:

1. If you are starting with a 16-bit Windows application, you must adapt your source code to the 32-bit environment.
2. You must compile the application using a 32-bit compiler.
3. You must link the application with the 32-bit libraries.
4. You must bind the 32-bit application with the 32-bit supervisor.
5. You can then run and/or debug the application.

20 Windows 3.x 32-bit Programming Overview

This chapter includes the following topics:

- WINDOWS.H
- Environment Notes
- Floating-point Emulation
- Multiple Instances
- Pointer Handling
- When To Convert Incoming Pointers
- When To Convert Outgoing Pointers
- SendMessage and SendDlgItemMessage
- GlobalAlloc and LocalAlloc
- Callback Function Pointers
- Window Sub-classing
- Calling 16-bit DLLs
- Making DLL Calls Transparent
- Far Pointer Manipulation
- _16 Functions

20.1 WINDOWS.H

When developing programs, make sure `WINDOWS.H` is included as the first include file in all source files. This header file contains only the following lines:

```
#ifdef _ WINDOWS_ 16_
#include <win16.h>
#else
#include <_ win386.h>
#endif
```

The file `WIN16.H` is the regular 16-bit Windows header file, and is only conditionally included for 16-bit Windows applications. The file `_ WIN386.H` contains all the prototypes and macros for the 32-bit environment, as well as including and modifying `WIN16.H.` These modifications are changing int to

short, and changing the ***far*** keyword to nothing. These changes (that ONLY apply to things defined in WIN16.H) cause all integers to be 16-bit integers, and all LP... pointer types to be near pointers.

Other include files for Windows must be specifically requested by defining macros before including WINDOWS.H. This is required so that the same changes made to the primary Windows header file will apply to routines declared in the other header files.

Macro name	*File included*
#define INCLUDE_COMMDLG_H	COMMDLG.H
#define INCLUDE_CUSTCNTL_H	CUSTCNTL.H
#define INCLUDE_DDE_H	DDE.H
#define INCLUDE_DDEML_H	DDEML.H
#define INCLUDE_DRIVINIT_H	DRIVINIT.H
#define INCLUDE_LZEXPAND_H	LZEXPAND.H
#define INCLUDE_MMSYSTEM_H	MMSYSTEM.H
#define INCLUDE_OLE_H	OLE.H
#define INCLUDE_PENWIN_H	PENWIN.H
#define INCLUDE_PENWOEM_H	PENWOEM.H
#define INCLUDE_PRINT_H	PRINT.H
#define INCLUDE_SHELLAPI_H	SHELLAPI.H
#define INCLUDE_STRESS_H	STRESS.H
#define INCLUDE_TOOLHELP_H	TOOLHELP.H
#define INCLUDE_VER_H	VER.H

20.2 Environment Notes

- The Windows functions ***Catch*** and ***Throw*** save only the 16-bit state. Instead of these functions, use the ***setjmp*** and ***longjmp*** functions.

- The 32-bit Windows Supervisor uses the first 256 bytes of the 32-bit application's stack to save state information. If this is corrupted, your application will abnormally terminate.

- The 32-bit Windows Supervisor provides resources for up to 512 callback routines. Note that this restriction is only on the maximum number of active callbacks.

20.3 Floating-point Emulation

The file WEMU387.386 is included to support floating-point emulation for 32-bit applications running under Windows. This file is installed in the [386Enh] section of your SYSTEM.INI file. By using the floating-point emulator, your application can be compiled with the "fpi87" option to use inline floating-point instructions, and it will run on a machine without a numeric coprocessor.

Only one of WEMU387.386 and WDEBUG.386 may be installed in your [386Enh] section. WEMU387.386 may be distributed with your application.

20.4 Multiple Instances

Since the 32-bit application resides in a flat memory space, it is NOT possible to share code with other instances. This means that you must register new window classes with callbacks into the new instance's code space. A simple way of accomplishing this is as follows:

```
int PASCAL WinMain( HANDLE hInstance,
                    HANDLE hPrevInstance;
                    LPSTR lpCmdLine,
                    int nCmdShow );
{
    WNDCLASS  wc;
    HWND      hWnd
    char      class[32];
    wc.style = NULL;
    wc.lpfnWndProc = (LPVOID) MainWndProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInstance;
    wc.hIcon = LoadIcon( NULL, IDI_APPLICATION );
    wc.hCursor = LoadCursor( NULL, IDC_ARROW );
    wc.hbrBackground = GetStockObject( WHITE_BRUSH );
    wc.lpszMenuName = "Menu";
    sprintf( class,"Class%d",hInstance );
    wc.lpszClassName = class;
    RegisterClass( &wc );
    hWnd = CreateWindow(
        class,
        "Application",
        WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        NULL,
        NULL,
        hInstance,
        NULL
    );
```

The variable ***class*** contains a unique name based on the instance of the application.

20.5 Pointer Handling

Windows 3.x is a 16-bit operating system. Function pointers that Windows deals with are 16-bit far pointers, and any data you communicate to Windows with are 16-bit far pointers. 16-bit far pointers occupy 4 bytes of data, and are capable of addressing up to 64K. For data objects larger than 64K, huge pointers are used (a sequence of far pointers that map out consecutive 64K segments for the data object). 16-bit far pointers are expensive to use due to the overhead of selector loads (each time you use the pointer, a segment register must have a value put in it). 16-bit huge pointers are even more expensive: not only is there the overhead of selector loads, but a run-time call is necessary to perform any pointer arithmetic.

In a 32-bit flat memory model, such as that of the Open Watcom C/C++ for Windows environment, all pointers are 32-bit near pointers (occupying 4 bytes of data as well). However, these pointers may access objects of up to 4 gigabytes in size, and there is no selector load overhead.

All Windows defined pointer types (e.g., LPSTR) are by default near pointers, not far pointers. To obtain a far pointer, the far keyword must be explicitly coded, i.e., `char far *foo`, rather than `LPSTR foo`. A 32-bit near pointer is the same size as a 16-bit far pointer, so that all Windows pointers are the same size in the 32-bit flat memory model as they are in the original 16-bit segmented model.

For a 32-bit environment to communicate with Windows 3.x, there are some considerations. All pointers sent to Windows must be converted from 32-bit near pointers to 16-bit far pointers. These conversions are handled by the Supervisor.

It is important to remember that all API functions which accept pointers (with the exception of functions that accept function pointers) accept 32-bit near pointers in this 32-bit model. If you attempt to pass a 32-bit far pointer, the conversion will not take place correctly.

16-bit far pointers to data may be passed into the API functions, and the Supervisor will not do any conversion.

Incoming pointers must be converted from 16-bit far pointers to 32-bit far pointers. This conversion is a trivial one: the offset portion of the 16-bit far pointer is extended to 32-bits. Pointers from Windows are by their nature far (that is, the data is pointed to by its own selector), and must be used as far in the 32-bit environment. Of course, conversions are only required if you actually need to reference the pointer.

Function pointers (i.e., pointers to callback routines) used by Windows are not converted from 32-bit to 16-bit. Rather, a 16-bit thunking layer that transfers control from the 16-bit environment to the 32-bit environment must be used. This thunking layer is provided by the Supervisor.

20.5.1 When To Convert Incoming Pointers

Whenever you wish to use a pointer passed to you by Windows, you must convert it to a 32-bit far pointer. If you are passed a 16-bit far pointer, the macro ***MK_FP32*** can be used to convert it to a 32-bit far pointer. If you are passed a 16-bit near pointer (e.g., from ***LocalLock***), then the macro ***MK_LOCAL32*** can be used to convert it to a 32-bit far pointer.

Some places where pointer conversion may be required are:

- LocalLock
- GlobalLock
- the lParam in a window callback routine (if it is a pointer)

20.5.2 When To Convert Outgoing Pointers

Typically, there is no need to do any kind of conversions on your pointers when passing them to Windows. The Supervisor handles all 32-bit to 16-bit translations for you, in the case of the regular Windows API functions. However, if you are passing a 32-bit pointer to some other 16-bit application in the Windows environment, then pointer conversions must by done. There are two types of "outgoing" pointers: data pointers and function pointers.

Function pointers (to callback routines) must have a thunking layer provided, using the ***GetProc16*** function (this is explained in detail in a later section).

Data pointers can be translated from 32-bit to 16-bit using the ***AllocAlias16*** and ***AllocHugeAlias16*** functions. These functions create 16-bit far pointers that have the same linear address as the 32-bit near pointer that was converted.

It is important to remember that when passing a pointer to a data structure in this fashion, any pointers in the data structure must also be converted.

The Supervisor will convert any pointers that it knows about; but there are some complications created by the fact that Windows allows you to pass pointers in functions that are prototyped to take a long integer.

The Windows API functions ***SendMessage*** and ***SendDlgItemMessage*** rely on other fields determining the nature of the long data item that they accept; this is discussed in detail in the next section.

20.5.2.1 SendMessage and SendDlgItemMessage

SendMessage and ***SendDlgItemMessage*** have special cover functions that determine when the long integer is really a pointer and needs to be converted. These cover functions are used automatically, unless the macro ***NOCOVERSENDS*** is defined before including `WINDOWS.H` as in the following example.

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

SendMessage and ***SendDlgItemMessage*** will do pointer conversions automatically using ***AllocAlias16*** and ***FreeAlias16*** (unless ***NOCOVERSENDS*** is defined) for the following message types:

- combo boxes (CB_ messages)
- edit controls (EM_ messages)
- list boxes (LB_ messages)
- certain windows messages (WM_ messages);

The messages that are intercepted by the cover functions for ***SendMessage*** and ***SendDlgItemMessage*** are:

```
CB_ADDSTRING        CB_DIR             CB_FINDSTRING
CB_FINDSTRINGEXACT  CB_GETLBTEXT        CB_INSERTSTRING
CB_SELECTSTRING

EM_GETLINE          EM_GETRECT         EM_REPLACESEL
EM_SETRECT          EM_SETRECTNP        EM_SETTABSTOPS

LB_ADDSTRING        LB_DIR             LB_FINDSTRING
LB_FINDSTRINGEXACT  LB_GETITEMRECT      LB_GETSELITEMS
LB_GETTEXT          LB_INSERTSTRING    LB_SELECTSTRING
LB_SETTABSTOPS

WM_MDICREATE        WM_NCCALCSIZE
```

Note that for ***SendMessage*** and ***SendDlgItemMessage***, some of the messages may NOT require pointer conversion:

- CB_ADDSTRING, CB_FINDSTRING, CB_FINDSTRINGEXACT, CB_INSERTSTRING will not need a conversion if the combo box was created as owner-draw style without CBS_HASSTRINGS style.

- LB_ADDSTRING, LB_FINDSTRING, LB_FINDSTRINGEXACT, LB_INSERTSTRING will not need a conversion if the list box was created as owner-draw style without LBS_HASSTRINGS style.

The macro ***NOCOVERSENDS*** should be defined in modules where messages like these are being sent. With these messages, the lParam data item does not contain a pointer, and the automatic pointer conversion would be incorrect. By doing

```
#define NOCOVERSENDS
#include "windows.h"
```

modules that send messages like the above will not have the pointer conversion performed.

20.5.3 GlobalAlloc and LocalAlloc

The functions ***GlobalAlloc*** and ***LocalAlloc*** are the typical way of allocating memory in the 16-bit Windows environment. In the 32-bit environment, there is no need to use these functions. The only time ***GlobalAlloc*** is needed is when allocating shared memory, i.e., ***GMEM_DDESHARE***.

The C runtime functions ***malloc*** and ***free*** manipulate your 32-bit near heap for you. By using these functions to allocate memory, you may create data objects as large as the enhanced mode Windows memory manager will permit.

20.5.4 Callback Function Pointers

To access a callback function, an instance of it must be created using ***MakeProcInstance***. This creates a "thunk" (a special piece of code) that automatically puts the application's data segment into the AX register, and then calls the specified callback function.

In Windows 3.x, it is not possible to do a ***MakeProcInstance*** directly on a 32-bit callback routine, since Windows 3.x does not understand 32-bit applications. Therefore, it is necessary to use a 16-bit callback routine that passes control to the 32-bit callback routine. This 16-bit callback routine is automatically created by the Supervisor when using any of the standard Windows API functions that accept a callback routine.

The 16-bit callback routine for a 32-bit application is a special layer that transfers the parameters from a 16-bit stack to the 32-bit stack, and then passes control to 32-bit code. These 16-bit callback routines are found in the Supervisor. The function ***GetProc16*** provides pointers to these 16-bit callback routines.

However, it is not often necessary to use the ***GetProc16*** function to obtain a 16-bit/32-bit callback interface function.

In the general case, one would have to write code as follows:

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

CALLBACKPTR     pCb;
FARPROC         fpProc;

pCb = GetProc16( A_ Function, GETPROC_ callbacktype );
fpProc = MakeProcInstance( pCb, hInstance );

/* do stuff */

Do_ it( ..., fpProc, ... );

/* do more stuff */

FreeProcInstance( fpProc );
ReleaseProc16( pCb );
```

It is not necessary to use this general code in the case of the regular Windows API functions. The following functions will automatically allocate the correct 16-bit/32-bit callback interface functions:

- ChooseColor
- ChooseFont
- CorrectWriting
- CreateDialog
- CreateDialogIndirect
- CreateDialogIndirectParam
- CreateDialogParam
- DdeInitialize
- DialogBox
- DialogBoxIndirect
- DialogBoxIndirectParam
- DialogBoxParam
- DictionarySearch
- EnumChildWindows
- EnumFontFamilies
- EnumFonts
- EnumMetaFile
- EnumObjects
- EnumProps
- EnumSymbols
- EnumTaskWindows
- EnumWindows
- Escape (SETABORTPROC option)
- FindText
- GetOpenFileName
- GetSaveFileName
- GlobalNotify
- GrayString
- LineDDA
- mciSetYieldProc
- mmioInstallIOProc
- NotifyRegister
- PrintDlg

- ProcessWriting
- Recognize
- RecognizeData
- RegisterClass
- ReplaceText
- SetClassLong (GCL_WNDPROC option)
- SetPenHook
- SetResourceHandler
- SetTimer
- SetWindowLong (GWL_WNDPROC option)
- SetWindowsHook
- SetWindowsHookEx
- TrainInk

As well, the following functions are covered to provide support for automatic creation of 16-bit callback routines:

- FreeProcInstance
- MakeProcInstance
- UnhookWindowsHook

If you need to get a callback that is not used by one of the above functions, then you must code the general case. Typically, this is required when a DLL needs a callback routine. In modules where this is necessary, you define the macro ***NOAUTOPROCS*** before you include `WINDOWS.H` as in the following example.

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

Be careful of the following when using ***NOAUTOPROCS***.

1. The call to ***MakeProcInstance*** and ***FreeProcInstance*** for the callback function occurs in a module with ***NOAUTOPROCS*** defined.

2. No Windows API functions (listed above) are used in the module with ***NOAUTOPROCS*** defined. If they are, you must code the general case to use them.

Note that ***NOAUTOPROCS*** is in effect on a module-to-module basis only.

You can avoid using ***NOAUTOPROCS*** on a call-by-call basis, if you do the following:

```
#undef <function>
<function>
Note: re-defining is only needed if you want to
      use a covered version of the function later on.
#define <function> _ Cover_ <function>
```

For example:

```
{
#undef SetWindowsHook
#undef MakeProcInstance

    FARPROC fp,oldfp;
    CALLBACKPTR cbp;
```

```
        cbp = GetProc16( CallbackHook, GETPROC_ CALLBACK );
        fp = MakeProcInstance( cbp, hInstance );
        oldfp = SetWindowsHook( WH_ CALLWNDPROC, fp );

}
```

This allows you to add general case code in the same module, without having to break the module into two parts.

RegisterClass automatically does a GetProc16 for the callback function, unless the macro NOCOVERRC is specified before including `WINDOWS.H` as in the following example.

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

20.5.4.1 Window Sub-classing

Sub-classing a Windows control in the 32-bit environment is straightforward. In fact, the code is identical to the code used in the 16-bit environment. A simple example is:

```
FARPROC fpOldProc;

long FAR PASCAL SubClassProc( HWND hWnd,
                              unsigned message,
                              WORD wParam,
                              LONG lParam )
{
    /*
     * code for sub-classing here
     */
    return( CallWindowProc( fpOldProc, hWnd, message,
                            wParam, lParam ) );
}

void SubClassDemo( void )
{
    HWND                hControl;
    FARPROC             fp;
    extern HANDLE       ProgramInstance;

    /* assume hControl gets created in here */

    fpOldProc = (FARPROC) GetWindowLong( hControl, GWL_ WNDPROC );
    fp = MakeProcInstance( SubClassProc, ProgramInstance );
    SetWindowLong( hControl, GWL_ WNDPROC, (LONG) fp );

    /* set it back */
    SetWindowLong( hControl, GWL_ WNDPROC, (LONG) fpOldProc );
    FreeProcInstance( fp );
}
```

Note that ***SetWindowLong*** is covered to recognize ***GWL_WNDPROC*** and automatically creates a 16-bit callback for the 32-bit callback. When replacing the callback routine with the original 16-bit routine, the covered version of ***SetWindowLong*** recognizes that the function is not a 32-bit callback, and so passes the pointer right through to Windows unchanged.

20.6 Calling 16-bit DLLs

A 16-bit function in a DLL can be called using the ***_Call16*** function. The first argument to ***_Call16*** is the address of the 16-bit function. This address is usually obtained by calling ***GetProcAddress*** with the name of the desired function. The second argument to ***_Call16*** is a string identifying the types of the parameters to be passed to the 16-bit function.

Character	*Parameter Type*
c	call a 'cdecl' function as opposed to a 'pascal' function (if specified, it must be listed first)
b	unsigned BYTE
w	16-bit WORD
d	32-bit DWORD
f	double precision floating-point
p	32-bit flat pointer (converted to 16:16 far pointer)

The 16-bit function must use either the ***PASCAL*** or ***CDECL*** calling convention. ***PASCAL*** calling convention is the default. If the function uses the ***CDECL*** calling convention, then you must specify the letter "c" as the first character of the argument type string.

Pointer types will automatically be converted from 32-bit near pointers to 16-bit far pointers before the function is invoked. Note that this pointer is only valid over the period of the call; after control returns to the 32-bit application, the 16-bit pointer created by the Supervisor is no longer valid.

The return value from ***_Call16*** is a ***DWORD***.

```
#include <windows.h>
HANDLE hDrv;
FARPROC lpfn;
int cb;
    if( (hDrv = LoadLibrary ("foo.dll")) < 32 )
        return FALSE;
    if( !(lpfn = GetProcAddress (hDrv, "ExtDeviceMode")) )
        return FALSE;
    /*
     * now, invoke the function
     */
    cb = (WORD) _Call16(
           lpfn,                // address of function
           "wwdppddw",          // parameter type info
           hwnd,                // parameters ...
           hDrv,
           NULL,
           "POSTSCRIPT PRINTER",
           "LPT1",
           NULL,
           NULL,
           0
            );
```

20.6.1 Making DLL Calls Transparent

This section gives an example of how to make your source code look as if you are calling the 16-bit DLL directly.

Assume there are 3 functions that you want to call in the 16-bit DLL, with prototypes as follows:

```
HWND FAR PASCAL Initialize( WORD start_ code );
BOOL FAR PASCAL DoStuff( HWND win_ hld, HDC win_ dc );
void FAR PASCAL Finish( void );
```

A fragment from the header file that you would include in your 32-bit application would be as follows:

```
extern FARPROC InitializeAddr;
extern FARPROC DoStuffAddr;
extern FARPROC FinishAddr;
#define Initialize( start_ code ) \
  _ Call16( InitializeAddr, "w", (WORD)start_ code )
#define DoStuff( win_ hld, data ) \
  _ Call16( DoStuffAddr, "wp", (HWND)win_ hld, (LPVOID)data )
#define Finish( void )  _ Call16( FinishAddr, "" )
```

The header file fragment gives external references for the function addresses for each function, and sets up macros do a ***_Call16*** for each of the functions.

At start up, you would call the following function:

```
/*
 * LoadDLL - get DLL ready for 32-bit use
 */
BOOL LoadDLL( void )
{
    HANDLE  dll;

    dll = LoadLibrary( "chart.dll" );
    if( dll < 32 ) return( FALSE);
    InitializeAddr = GetProcAddress( dll, "Initialize" );
    DoStuffAddr = GetProcAddress( dll, "DoStuff" );
    FinishAddr = GetProcAddress( dll, "Finish" );
    return( TRUE );
}
```

This function loads the 16-bit DLL and gets the addresses for all of the entry points in the DLL. By including the header file with all the macros in it, you can code calls to the DLL functions as if you were calling the functions directly. For example:

```
#include <windows.h>
#include "fragment.h"
char *data = "the data";

void TestDLL( void )
{
    HWND res;
```

```
    if( !LoadDLL() ) {
        MessageBox( NULL, "Could not load DLL",
                    "Error", MB_ OK );
        return;
    }
    res = Initialize( 1 );
    DoStuff( res, data );
    Finish();
}
```

20.7 Far Pointer Manipulation

The following C library functions are available for manipulating far data. These are useful when using pointers obtained by ***MK_FP32*** and ***MK_LOCAL32***.

Memory manipulation:

- _fmemccpy
- _fmemchr
- _fmemcmp
- _fmemcpy
- _fmemicmp
- _fmemmove
- _fmemset

String manipulation:

- _fstrcat
- _fstrchr
- _fstrcmp
- _fstrcpy
- _fstrcspn
- _fstricmp
- _fstrlen
- _fstrlwr
- _fstrncat
- _fstrncmp
- _fstrncpy
- _fstrnicmp
- _fstrnset
- _fstrpbrk
- _fstrrchr
- _fstrrev
- _fstrset
- _fstrspn
- _fstrtok
- _fstrupr

20.8 _16 Functions

Every Windows API function that accepts a pointer has a corresponding _16 function. The _16 version of the function will not convert any of the pointers that it accepts; it will assume that all pointers are 16-bit far pointers already. This applies to both data and function pointers.

Some sample code demonstrating the use for this is:

```
void ReadEditBuffer( char *fname, HWND hwndEdit )
{
    int       file;
    HANDLE    hText;
    char far *flpData;
    LPSTR     lpData;
    WORD      filelen;
    /*
     * no error checking is performed; we just
     * assume everything works for this example.
     */
    file = _ lopen( fname, 0);
    filelen = _ llseek( file, 0L, 2 );

    hText = (HANDLE) SendMessage( hwndEdit, EM_ GETHANDLE,
                                  0, 0L );
    LocalReAlloc( hText, filelen+1, LHND );
    flpData = MK_ LOCAL32( LocalLock( hText ) );
    lpData = (LPSTR) MK_ FP16( flpB );
    _ 16_ lread( file, lpData, filelen );
    _ lclose( file );

}
```

This example reads the contents of a file into the buffer of an edit window. Because the edit window's memory is located in the local heap, which is the Supervisor's heap, the ***MK_LOCAL32*** function is needed to access the data. The ***MK_FP16*** macro compresses the 32-bit far pointer into a 16-bit far pointer, which can then be used by the _16_lread function.

21 Windows 32-Bit Dynamic Link Libraries

21.1 Introduction to 32-Bit DLLs

Open Watcom C/C++ allows the creation of 32-bit Dynamic Link Libraries (DLL). In fact, 32-bit DLLs are simpler to write than 16-bit DLLs. A 16-bit DLL runs on the caller's stack, and thus DS != SS. This creates difficulties in the small and medium memory models because near pointers to local variables are different from near pointers to global variables. The 32-bit DLL runs on its own stack, in the usual flat memory space, which eliminates these concerns.

There is a special version of the supervisor, `W386DLL.EXT` that performs a similar job to `WIN386.EXT`. However, the 32-bit DLL supervisor is a 16-bit Windows DLL, rather than a 16-bit Windows application. On the first use of the 32-bit DLL, the DLL supervisor loads the 32-bit DLL and invokes the 32-bit initialization routine (the DLL's `WinMain` routine). The initialization routine declares all entry points (via `DefineDLLEntry`) and performs any other necessary initialization. An index number in the range 1 to 128 is used to identify all external 32-bit DLL routines. `DefineDLLEntry` is used to assign an index number to each routine, as well as to identify the arguments.

The DLL supervisor contains a general entry point for Windows applications to call into called `Win386LibEntry`. It also contains 128 specific entry points called `DLL1` to `DLL128` which correspond to the entry points established via `DefineDLLEntry` (the first argument to `DefineDLLEntry` is an index number in the range 1 to 128). These entry points are `FAR PASCAL` functions. All applications call into the 32-bit DLL via these entry points. They build the necessary stack frame and switch to the 32-bit DLL's data space.

If you call via `Win386LibEntry` then you pass the DLL entry point number or index (1 to 128) as the last argument. `Win386LibEntry` uses this index number to call the appropriate 32-bit DLL routine. From a pseudo-code point of view, the 16-bit supervisor might look like the following:

```
DLL1::  set index=1
        invoke 32bitDLLindirect

DLL2::  set index=2
        invoke 32bitDLLindirect
            .
            .
            .
DLL128:: set index=128
         invoke 32bitDLLindirect

Win386LibEntry::
        set index from index_ argument
        invoke 32bitDLLindirect

32bitDLLindirect:
        set up stack frame
        switch to 32-bit data space
        call indirect registration_ list[ index ]
            .
            .
            .
```

When you are creating a 32-bit DLL, keep in mind that the entry points you define may be invoked by a 16-bit application as well as a 32-bit application. It is for this reason that all far pointers passed to a 32-bit DLL are 16-bit far pointers. Hence, whenever a pointer is passed as an argument to a 32-bit DLL entry point and you wish to access the data it points to, you must convert the pointer appropriately.

21.2 A Sample 32-bit DLL

Let us begin our discussion of DLLs by showing the code for a simple DLL. The source code for these examples is provided in the `\WATCOM\SAMPLES\DLL` directory. We describe how to compile and link the examples in the section entitled "Creating and Debugging Dynamic Link Libraries" on page 162. The code for this DLL can be compiled with the 16-bit compiler to produce a 16-bit DLL and it can can be compiled with the 32-bit compiler to produce a 32-bit DLL. The example illustrates the fundamental differences between the two types of DLLs. The 32-bit DLL has a `WinMain` routine and the 16-bit DLL has a `LibMain` routine.

Example:

```
/*
 *  DLL.C
 */
#include <stdio.h>
#include <windows.h>

#if defined(__386__)/* if we are doing a 32-bit DLL */
  #define DLL_ID "DLL32"
#else               /* else we are doing a 16-bit DLL */
  #define DLL_ID "DLL16"
#endif
```

```
long FAR PASCAL _ _ export FooMe1(WORD w1, DWORD w2, WORD w3)
{
  char buff[128];

  sprintf( buff, "FooMe1: w1=%hx, w2=%lx, w3=%hx",
           w1, w2, w3 );
  MessageBox( NULL, buff, DLL_ ID, MB_ OK );
  return( w1 + w2 );
}

long FAR PASCAL _ _ export FooMe2( DWORD w1, WORD w2 )
{
  char buff[128];

  sprintf( buff, "FooMe2: w1=%lx, w2=%hx", w1, w2 );
  MessageBox( NULL, buff, DLL_ ID, MB_ OK );
  return( w1 + 1 );
}

#if defined(_ _ 386_ _ )/* if we are doing a 32-bit DLL */
long PASCAL WinMain( HANDLE hInstance,
                     HANDLE hPrevInstance,
                     LPSTR  lpszCmdLine,
                     int    nCmdShow )
{
  if( DefineDLLEntry( 1, (void *) FooMe1, DLL_ WORD,
              DLL_ DWORD, DLL_ WORD, DLL_ ENDLIST )) {
      return( 0 );
  }
  if( DefineDLLEntry( 2, (void *) FooMe2, DLL_ DWORD,
              DLL_ WORD, DLL_ ENDLIST ) ) {
      return( 0 );
  }
  MessageBox( NULL, "32-bit DLL Started", DLL_ ID, MB_ OK );
  return( 1 );
}
#else               /* else we are doing a 16-bit DLL */
BOOL FAR PASCAL LibMain( HANDLE hInstance,
                         WORD wDataSegment,
                         WORD wHeapSize,
                         LPSTR lpszCmdLine )
{
  #if 0
  /*
    We can't use MessageBox here since static binding is
    used and a message queue has not been created by the
    time DLL16 is loaded.
  */
  MessageBox( NULL, "16-bit DLL Started", DLL_ ID, MB_ OK );
  #endif
  return( TRUE );
}
#endif
```

To create a 16-bit DLL from this code, the following steps must be performed.

Example:

```
C>wcc dll /mc /bt=windows /zu /fo=dll16
C>wlink system windows_ dll file dll16
C>wlib -n dll16 +dll16.dll
```

To create a 32-bit DLL from this code, the following steps must be performed.

Example:

```
C>wcc386 dll /bt=windows /fo=dll32
C>wlink system win386 file dll32
C>wbind -n -d dll32
```

There are two entry points defined, `FooMe1` (index number 1) and `FooMe2` (index number 2). `FooMe1` accepts three arguments: a WORD, a DWORD, and a WORD. `FooMe2` accepts two arguments: a DWORD and a WORD.

`WinMain` returns zero to notify Windows that the DLL initialization failed, and returns a one if initialization succeeds.

`WinMain` accepts the same arguments as the `WinMain` procedure of a regular Windows program, however, only two arguments are used. `hInstance` is the DLL handle and `lpszCmdLine` is the command line passed to the DLL.

21.3 Calling Functions in a 32-bit DLL from a 16-bit Application

The following is a 16-bit Windows program that demonstrates how to call the two routines defined in our DLL example.

Example:

```
/*
 *  EXE16.C
 */
#include <stdio.h>
#include <windows.h>

#define Add3 1
#define Add2 2

typedef long (FAR PASCAL *FPROC)();
typedef long (FAR PASCAL *FARPROC1)(WORD, DWORD, WORD, int);
typedef long (FAR PASCAL *FARPROC2)(DWORD, WORD, int);

long FAR PASCAL FooMe1( WORD, DWORD, WORD );
long FAR PASCAL FooMe2( DWORD, WORD );
```

```
int PASCAL WinMain( HANDLE hInstance,
                    HANDLE hPrevInstance,
                    LPSTR  lpszCmdLine,
                    int    nCmdShow )
{
  FPROC fp;
  HANDLE hlib;
  long cb;
  char buff[128];

  MessageBox( NULL, "16-bit EXE Started", "EXE16", MB_OK );

  /* Do the 16-bit demo using static binding */
  cb = FooMe1( 0x666, 0x77777111, 0x6969 );
  sprintf( buff, "RC1 = %lx", cb );
  MessageBox( NULL, buff, "EXE16", MB_OK );

  cb = FooMe2( 0x12345678, 0x8888 );
  sprintf( buff, "RC2 = %lx", cb );
  MessageBox( NULL, buff, "EXE16", MB_OK );

  /* Do the 32-bit demo */
  hlib = LoadLibrary( "dll32.dll" );
  fp = (FPROC) GetProcAddress( hlib, "Win386LibEntry" );

  cb = (*(FARPROC1)fp)( 0x666, 0x77777111, 0x6969, Add3 );
  sprintf( buff, "RC1 = %lx", cb );
  MessageBox( NULL, buff, "EXE16", MB_OK );

  cb = (*(FARPROC2)fp)( 0x12345678, 0x8888, Add2 );
  sprintf( buff, "RC2 = %lx", cb );
  MessageBox( NULL, buff, "EXE16", MB_OK );

  return( 0 );
}
```

Note that the last argument of a call to the 32-bit DLL routine is the index number of the 32-bit DLL routine to use. To create the 16-bit sample Windows executable from this code, the following steps must be performed.

Example:

```
C>wcc exe16 /bt=windows
C>wlink system windows file exe16 library dll16
```

21.4 Writing a 16-bit Cover for the 32-bit DLL

The following is a suggested way to make a 32-bit DLL behave just like a 16-bit DLL from the point of view of the person trying to use the DLL.

Create a library of cover functions for each of the entry points. Each library entry would call the 32-bit DLL using the appropriate index number.

For example, assume we have 3 functions in our DLL, `Initialize`, `DoStuff`, and `Finish`. Assume `Initialize` takes an integer, `DoStuff` takes an integer and a pointer, and `Finish` takes nothing. We could build a 16-bit library as follows:

Example:

```
#include <windows.h>
typedef long (FAR PASCAL *FPROC)();
extern long FAR PASCAL Win386LibEntry();
FPROC LibEntry = Win386LibEntry;

BOOL Initialize( int parm )
{
    return( LibEntry( parm, 1 ) );
}

int DoStuff( int parm1, LPVOID parm2 )
{
    return( LibEntry( parm1, parm2, 2 ) );
}

void Finish( void )
{
    LibEntry( 3 );
}
```

21.5 Creating and Debugging Dynamic Link Libraries

In the following sections, we will take you through the steps of compiling, linking, and debugging both 16-bit and 32-bit Dynamic Link Libraries (DLLs).

We will use example programs that are provided in source-code form in the Open Watcom C/C++ package. The files described in this chapter are located in the directory `\WATCOM\SAMPLES\DLL`. The following files are provided:

GEN16.C	is the source code for a generic 16-bit Windows application that calls functions in a 32-bit Windows DLL.
GEN16.LNK	is the linker directive file for linking the 16-bit Windows application.
GEN32.C	is the source code for a generic 32-bit Windows application that calls functions in both 16-bit and 32-bit Windows DLLs.
GEN32.LNK	is the linker directive file for linking the 32-bit Windows application.
DLL16.C	is the source code for a simple 16-bit DLL containing one library routine.
DLL16.LNK	is the linker directive file for linking the 16-bit Windows DLL.
DLL32.C	is the source code for a more complex 32-bit DLL containing three library routines.
DLL32.LNK	is the linker directive file for linking the 32-bit Windows DLL.

EXE16.C	is the source code for a generic 16-bit Windows application that calls functions in both 16-bit and 32-bit Windows DLLs.
DLL.C	is the source code for a DLL containing three library routines. The source code for this DLL can be used to create both 16-bit and 32-bit DLLs.
MAKEFILE	is a makefile for compiling and linking the programs described above.

21.5.1 Building the Applications

To create the DLLs and test applications, we will use the WATCOM Open Watcom Make utility and the supplied makefile.

Example:

```
C>wmake -f makefile
```

21.5.2 Installing the Examples under Windows

Start up Microsoft Windows 3.x if you have not already done so. Add the `EXE16.EXE` file to one of your Window groups using the Microsoft Program Manager.

1. Select the "New..." entry from the "File" menu of the Microsoft Windows Program Manager.

2. Select "Program Item" from the "New Program Object" window and press the "OK" button.

3. Enter "DLL Test" as a description for the EXE16 program. Enter the full path to the EXE16 program as a command line.

   Example:

   ```
   Description:    Test
   Command Line:   c:\work\dll\exe16.exe
   ```

21.5.3 Running the Examples

Start the 16-bit application by double clicking on its icon. A number of message boxes are presented. You may wish to compare the output in each message box with the source code of the program to determine if the correct results are being obtained. Click on the "OK" button as each of them are displayed.

21.5.4 Debugging a 32-bit DLL

The Open Watcom Debugger can be used to debug a DLL. To debug a 32-bit DLL, a "breakpoint" instruction must be inserted into the source code for the DLL at the "WinMain" entry point. This is done using the "pragma" compiler directive. We have already added the breakpoint to the source code for the 32-bit DLL.

Example:

```
extern void BreakPoint( void );
#pragma aux BreakPoint = 0xcc;

int PASCAL WinMain( HANDLE hInstance,
                    HANDLE x1,
                    LPSTR lpCmdLine,
                    int x2 )
{

  BreakPoint();
  DefineDLLEntry( 1, (void *) Lib1,
                        DLL_ WORD,
                        DLL_ DWORD,
                        DLL_ WORD,
                .
                .
                .
```

Start up Microsoft Windows 3.x if you have not already done so. Start the debugger by double-clicking on the Open Watcom Debugger icon. At the prompt, enter the path specification for the application. When the debugger has successfully loaded EXE16, start execution of the program. When the breakpoint is encountered in the 32-bit DLL, the debugger is re-entered. The debugger will automatically skip past the breakpoint.

From this point on, you can symbolically debug the 32-bit DLL. You might, for example, set breakpoints at the start of each DLL routine to debug each of them as they are called.

21.5.5 Summary

Note that the "WinMain" entry point is only called once, at the start of any application requesting it. After this, the "WinMain" entry point is no longer called. You may have to restart Windows to debug this section of code a second or third time.

22 Interfacing Visual Basic and Open Watcom C/C++ DLLs

22.1 Introduction to Visual Basic and DLLs

This chapter describes how to interface Microsoft Visual Basic 3.0 applications and 32-bit Dynamic Link Libraries (DLLs) created by Open Watcom C/C++. It describes how to write functions for a 32-bit DLL, how to compile and link them, and how to call these functions from Visual Basic. One of the proposed techniques involves the use of a set of cover functions in a 16-bit DLL so, indirectly, this chapter also describes interfacing to 16-bit DLLs.

It is possible to invoke the `Win386LibEntry` function (Open Watcom's 32-bit function entry point, described below) directly from Visual Basic. However, this technique limits the arguments that can be passed to a 32-bit DLL. The procedure and problems are explained below.

To work around the problem, a 16-bit DLL can be created, that covers the 32-bit DLL. Within the 16-bit DLL, we will place cover functions that will call the corresponding 32-bit function in the 32-bit DLL. We illustrate the creation of the 16-bit DLL using the 16-bit C compiler in Open Watcom C/C++.

Before we begin our example, there are some important technical issues to consider.

The discussion in this chapter assumes that you, the developer, have a working knowledge of Visual Basic, including how to bring up the general declarations screen, how to create command buttons, and how to associate code with command buttons. You must use Visual Basic 3.0 or later. Visual Basic Version 2.x will not work because of a deficiency in this product regarding the calling of functions in DLLs.

For the purposes of the following discussion, you should have installed both the 16-bit and 32-bit versions of Open Watcom C/C++, as well as version 3.0 or later of Visual Basic. Ensure that the **PATH, INCLUDE** and **WINDOWS_INCLUDE** environment variables are defined to include at least the directories indicated. We have assumed that Open Watcom C/C++ is installed in the `c:\watcom` directory, and Visual Basic is in the `c:\vb` directory:

```
set path=c:\watcom\binw;c:\vb;c:\dos;c:\windows
set include=c:\watcom\h
set windows_ include=c:\watcom\h\win
```

Open Watcom's 32-bit DLL supervisor contains a general entry point for Windows applications to call into called `Win386LibEntry.` It also contains 128 specific entry points called `DLL1` to `DLL128` which correspond to the entry points established via `DefineDLLEntry` (the first argument to `DefineDLLEntry` is an index number in the range 1 to 128). All applications call into the 32-bit DLL via these entry points. They build the necessary stack frame and switch to the 32-bit DLL's data space.

If you call via `Win386LibEntry` then you pass the DLL entry point number or index (1 to 128) as the last argument. `Win386LibEntry` uses this index number to call the appropriate 32-bit DLL routine.

In many languages and programs (such as C and Microsoft Excel), function calls are very flexible. In other words, a function can be called with different argument types each time. This is generally necessary for calling `Win386LibEntry` in a 32-bit extended DLL function. The reason is that this function takes the

same arguments as the function being called, as well as the index number of the called function. After the 32-bit flat model has been set up, `Win386LibEntry` then calls this function. In Visual Basic, once a function is declared as having certain arguments, it cannot be redeclared. For example, suppose we have a declaration as follows:

Example:

```
Declare Function Win386LibEntry Lib "c:\path\vbdll32.dll"
=> (ByVal v1 As Integer, ByVal v2 As Long, ByVal
=> v3 As Integer, ByVal I As Integer) As Long
```

(Note: the => means to continue the statement on the same line.) In this example, we could only call a function in any 32-bit extended DLL with a 16-bit integer as the first and third argument, and a 32-bit integer as the second argument. There are three ways to work around this deficiency in Visual Basic:

1. Use the Visual Basic "Alias" attribute to declare `Win386LibEntry` differently for each DLL routine. Reference the different DLL routines using these aliases.

2. Use the specific entry point, one of `DLL1` through `DLL128,` corresponding to the DLL routine that you want to call. Each entry point can be described to take different arguments. We can still use the "Alias" attribute to make the link between the name we use in the Visual Basic function and the name in the 32-bit extended DLL. This is the method that we will use in the "Direct Call" technique discussed below. It is simpler to use since it requires one less argument (you don't require the index number).

3. Use a method which involves calling functions in a 16-bit "cover" DLL written in a flexible-argument language, which then calls the functions in the 32-bit DLL. This is the "Indirect Call" method discussed below.

22.2 A Working Example

The best way to demonstrate these techniques is through an example. This example consists of a Visual Basic application with 3 push buttons. The first push button invokes a direct call to a 32-bit DLL which will display a message window with its arguments, the second push button invokes an indirect call to the same function through a 16-bit DLL, and the third button exits the Visual Basic application.

To create a Visual Basic application:

(1) **Start up a new project folder** from the "File" menu.

(2) **Select "View Form"** from the "Project" window.

(3) **Draw three command buttons** on the form by selecting command buttons from the "Toolbox" window.

(4) **Change the caption on each button.** To do this, highlight the first button. Then, open the "Properties" window. Double click on the "Caption window", and change the caption to "Direct call". Highlight the second button, and change its caption to "Indirect call". Highlight the third, changing the caption to "Exit".

Now, your Visual Basic application should have three push buttons, "Direct call", "Indirect call", and "Exit".

(5) Double click on the "Direct Call" button.

An edit window will pop up. Enter the following code:

```
Sub Command1_ Click ()
    Dim var1, var2 As Integer
    Dim varlong, worked As Long

    var1 = 230
    varlong = 215
    var2 = 32
    worked = Add3(var1, varlong, var2)
    Print worked
    worked = Add2(varlong, var2)
    Print worked
End Sub
```

(6) Double click on the "Indirect Call" button.

Another edit window will pop up. Enter the following code:

```
Sub Command2_ Click ()
    Dim var1, var2 As Integer
    Dim varlong, worked As Long

    var1 = 230
    varlong = 215
    var2 = 32
    worked = Function1( var1, varlong, var2 )
    Print worked
    worked = Function2( varlong, var2 )
    Print worked
End Sub
```

(7) Double click on the "Exit" command button and enter the following code in the pop-up window:

```
Sub Command3_ Click ()
    End
End Sub
```

(8) Select "View Code" from the "Project" window. To interface these Visual Basic functions to the DLLs, the following code is needed in the

```
Object: [general] Proc: [declarations]
```

section of the code. This code assumes that `VBDLL32.DLL` and `COVER16.DLL` are in the `c:\path` directory. Modify the pathnames appropriately if this is not the case. (Note: the => means to continue the statement on the same line.)

```
Declare Function Function1 Lib "c:\path\cover16.dll"
=> (ByVal v1 As Integer, ByVal v2 As Long,
=> ByVal v3 As Integer) As Long

Declare Function Function2 Lib "c:\path\cover16.dll"
=> (ByVal v1 As Long, ByVal v2 As Integer) As Long

Declare Function Add3 Lib "c:\path\vbdll32.dll"
=> Alias "DLL1"
=> (ByVal v1 As Integer, ByVal v2 As Long,
=> ByVal v3 As Integer) As Long

Declare Function Add2 Lib "c:\path\vbdll32.dll"
=> Alias "DLL2"
=> (ByVal v1 As Long, ByVal v2 As Integer) As Long
```

Now, when all of the code below is compiled correctly, and the Visual Basic program is run, the "Direct call" button will call the `DLL1` and `DLL2` functions directly, aliased as the functions `Add3` and `Add2` respectively. The "Indirect call" button will call the 16-bit DLL, which will then call the 32-bit DLL, for both `Function1` and `Function2`. To run the Visual Basic program, select "Start" from the "Run" menu.

22.3 Sample Visual Basic DLL Programs

The sample programs provided below are for a 32-bit DLL, and a 16-bit cover DLL, which will call the two functions contained in the 32-bit DLL.

22.3.1 Source Code for VBDLL32.DLL

```
/*
 *  VBDLL32.C
 */
#include <stdio.h>
#include <windows.h>    /* required for all Windows applications */

long FAR PASCAL Add3( short var1, long varlong, short var2 )
{
  char buf[128];

  sprintf( buf, "Add3: var1=%d, varlong=%ld, var2=%d",
               var1, varlong, var2 );
  MessageBox( NULL, buf, "VBDLL32", MB_ OK | MB_ TASKMODAL );
  return( var1 + varlong + var2 );
}

long FAR PASCAL Add2( long varlong, short var2 )
{
  char buf[128];

  sprintf( buf, "Add2: varlong=%ld, var2=%d", varlong, var2 );
  MessageBox( NULL, buf, "VBDLL32", MB_ OK | MB_ TASKMODAL );
  return( varlong + var2 );
}
```

```
#pragma off (unreferenced);
int PASCAL WinMain(HANDLE hInstance, HANDLE x1, LPSTR lpCmdLine, int x2)
#pragma on (unreferenced);
{
  DefineDLLEntry( 1, (void *) Add3, DLL_ WORD, DLL_ DWORD, DLL_ WORD,
                     DLL_ ENDLIST );
  DefineDLLEntry( 2, (void *) Add2, DLL_ DWORD, DLL_ WORD, DLL_ ENDLIST );
  return( 1 );
}
```

22.3.2 Source code for COVER16.DLL

The functions in this 16-bit DLL will call the functions in the 32-bit DLL, `VBDLL32.DLL`, shown above, with the appropriate `Win386LibEntry` call for each function.

```
/*
 *  COVER16.C
 */

#include <stdio.h>
#include <windows.h>    /* required for all Windows applications */

typedef long (FAR PASCAL *FPROC)();

FPROC DLL_ 1;
FPROC DLL_ 2;

long FAR PASCAL _ _ export Function1( short var1,
                                      long var2,
                                      short var3 )
{
    return( (long) DLL_ 1( var1, var2, var3 ) );
}

long FAR PASCAL _ _ export Function2( long var1, short var2 )
{
    return( (long) DLL_ 2( var1, var2 ) );
}

#pragma off (unreferenced);
BOOL FAR PASCAL LibMain( HANDLE hInstance, WORD wDataSegment,
                         WORD wHeapSize, LPSTR lpszCmdLine )
#pragma on (unreferenced);
{
    HANDLE hlib;

    /* Do our DLL initialization */
    hlib = LoadLibrary( "vbdll32.dll" );
    if( hlib < 32 ) {
        MessageBox( NULL,
                    "Make sure your PATH contains VBDLL32.DLL",
                    "COVER16", MB_ OK | MB_ ICONEXCLAMATION );
        return( FALSE );
    }
    DLL_ 1 = (FPROC) GetProcAddress( hlib, "DLL1" );
    DLL_ 2 = (FPROC) GetProcAddress( hlib, "DLL2" );
    return( TRUE );
}
```

22.4 Compiling and Linking the Examples

To create the 32-bit DLL `VBDLL32.DLL`, type the following at the command line (make sure that `VBDLL32.C` is in your current directory):

```
wcl386 vbdll32 -bt=windows -bd -d2 -l=win386
wbind vbdll32 -d -n
```

To create the 16-bit DLL `COVER16.DLL`, type the following at the command line (make sure that `COVER16.C` are in your current directory):

```
wcl cover16 -mc -bt=windows -bd -zu -d2 -l=windows_dll
```

Notes:

1. The "mc" option selects the compact memory model (small code, big data). The code for 16-bit DLLs must be compiled with one of the big data models.

2. The "bd" option indicates that a DLL will be created from the object files.

3. The "bt" option selects the "windows" target. This option causes the C or C++ compiler to generate Windows prologue/epilogue code sequences which are required for Microsoft Windows applications. It also causes the compiler to use the **WINDOWS_INCLUDE** environment variable for header file searches. It also causes the compiler to define the macro `__WINDOWS__` and, for the 32-bit C or C++ compiler only, the macro `__WINDOWS_386__`.

4. The "zu" option is used when compiling 16-bit code that is to be placed in a Dynamic Link Library (DLL) since the SS register points to the stack segment of the calling application upon entry to the function.

5. The "d2" option is used to disable optimizations and include debugging information in the object file and DLL. The techniques for debugging DLLs are described in the chapter entitled "Windows 32-Bit Dynamic Link Libraries" on page 157.

You are now ready to run the Visual Basic application.

23 WIN386 Library Functions and Macros

Each special Windows function or macro in the Open Watcom C/C++ library is described in this chapter. Each description consists of a number of subsections:

Synopsis: This subsection gives the header files that should be included within a source file that references the function or macro. It also shows an appropriate declaration for the function or for a function that could be substituted for a macro. This declaration is not included in your program; only the header file(s) should be included.

When a pointer argument is passed to a function and that function does not modify the item indicated by that pointer, the argument is shown with `const` before the argument. For example,

```
const char *string
```

indicates that the array pointed at by *string* is not changed.

Description: This subsection is a description of the function or macro.

Returns: This subsection describes the return value (if any) for the function or macro.

Errors: This subsection describes the possible `errno` values.

See Also: This optional subsection provides a list of related functions or macros.

Example: This optional subsection consists of one or more examples of the use of the function. The examples are often just fragments of code (not complete programs) for illustration purposes.

Classification: This subsection provides an indication of where the function or macro is commonly found. The functions or macros in this section are all classified as "WIN386" (i.e., they pertain to 32-bit Windows programming).

Synopsis:
```
#include <windows.h>
DWORD AllocAlias16( void *ptr );
```

Description: The AllocAlias16 function obtains a 16-bit far pointer equivalent of a 32-bit near pointer. These pointers are used when passing data pointers to Windows through functions that have DWORD arguments, and for any pointers within data structures passed this way.

Returns: The AllocAlias16 function returns a 16-bit far pointer usable by Windows, or returns 0 if the alias cannot be allocated.

See Also: FreeAlias16

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

  DWORD mcs_16;
  /*
   * Send a message to a MDI client to create a window.
   * _16SendMessage is used for this example, since it will
   * not do any pointer conversions automatically.
   */
  MDICREATESTRUCT mcs;
  mcs.szTitle = (LPSTR) AllocAlias16( "c:\\foo.bar" );
  mcs.szClass = (LPSTR) AllocAlias16( "mdichild" );
  mcs.hOwner  = hInst;
  mcs.x = mcs.cx = (int) CW_USEDEFAULT;
  mcs.y = mcs.cy = (int) CW_USEDEFAULT;
  mcs.style = 0;

  /* tell the MDI Client to create the child */
  mcs_16 = AllocAlias16( &mcs );
  hwnd = (WORD) _16SendMessage( hwndMDIClient,
                                WM_MDICREATE,
                                0,
                                (LONG) mcs_16 );
  FreeAlias16( mcs_16 );
  FreeAlias16( (DWORD) mcs.szClass );
  FreeAlias16( (DWORD) mcs.szTitle );
```

Classification: WIN386

Synopsis:
```
#include <windows.h>
DWORD AllocHugeAlias16( void *ptr, DWORD size );
```

Description: The AllocHugeAlias16 function obtains a 16-bit far pointer to a 32-bit memory object that is *size* bytes in size. This is similar to the function `AllocAlias16`, except that `AllocAlias16` will only give 16-bit far pointers to 32-bit memory objects of up to 64K in size. To get 16-bit far pointers to 32-bit memory objects larger than 64K, `AllocHugeAlias16` should be used.

Returns: The AllocHugeAlias16 function returns a 16-bit far pointer usable by Windows, or returns 0 if the alias cannot be allocated.

See Also: `AllocAlias16`, `FreeAlias16`, `FreeHugeAlias16`

Example:
```
#include <windows.h>
#include <malloc.h>
#define SIZE 300000

  DWORD alias;
  void *tmp;

  tmp = malloc( SIZE );
  alias = AllocHugeAlias16( tmp, SIZE );

  /* Windows calls using the alias ... */

  FreeHugeAlias16( alias, SIZE );
```

Classification: WIN386

Synopsis:

```
#include <windows.h>
DWORD _Call16( FARPROC lpFunc, char *fmt, ... );
```

Description: The _Call16 function performs the same function as `GetIndirectFunctionHandle`, `InvokeIndirectFunctionHandle` and `FreeIndirectFunctionHandle` but is much easier to use. The first argument *lpFunc* is the address of the 16-bit function to be called. This address is usually obtained by calling `GetProcAddress` with the name of the desired function. The second argument `fmt` is a string identifying the types of the parameters to be passed to the 16-bit function.

Character	*Parameter Type*
c	call a 'cdecl' function as opposed to a 'pascal' function (if specified, it must be listed first)
b	unsigned BYTE
w	16-bit WORD
d	32-bit DWORD
f	double precision floating-point
p	32-bit flat pointer (converted to 16:16 far pointer)

The 16-bit function must use either the `PASCAL` or `CDECL` calling convention. `PASCAL` calling convention is the default. If the function uses the `CDECL` calling convention, then you must specify the letter "c" as the first character of the argument type string.

Pointer types will automatically be converted from 32-bit near pointers to 16-bit far pointers before the function is invoked. Note that this pointer is only valid over the period of the call; after control returns to the 32-bit application, the 16-bit pointer created by the Supervisor is no longer valid.

Returns: The _Call16 function returns a 32-bit DWORD which represents the return value from the 16-bit function that was called.

See Also: `GetIndirectFunctionHandle`, `FreeIndirectFunctionHandle`

Example:

```
#include <windows.h>
HANDLE hDrv;
FARPROC lpfn;
int cb;

    if( (hDrv = LoadLibrary ("foo.dll")) < 32 )
        return FALSE;
    if( !(lpfn = GetProcAddress (hDrv, "ExtDeviceMode")) )
        return FALSE;

    /*
     * now, invoke the function
     */
    cb = (WORD) _Call16(
           lpfn,                // address of function
           "wwdppddw",          // parameter type info
           hwnd,                // parameters ...
           hDrv,
           NULL,
           "POSTSCRIPT PRINTER",
           "LPT1",
           NULL,
           NULL,
           0
            );
```

Classification: WIN386

Synopsis:
```
#include <windows.h>
int DefineDLLEntry( int index, void * routine, ... );
```

Description: The DefineDLLEntry function defines an *index* number for the 32-bit DLL procedure *routine*. The parameter *index* defines the index number that must be used in order to invoke the 32-bit FAR procedure *routine*. The variable argument list defines the types of parameters that will be received by the 32-bit DLL *routine*. Valid parameter types are:

DLL_PTR	16-bit far pointer
DLL_DWORD	32-bits
DLL_WORD	16-bits
DLL_CHAR	8-bits
DLL_ENDLIST	Marks the end of the variable argument list.

Note that all pointers are received as 16-bit far pointers. To access the data from the 32-bit DLL, the `MK_FP32` macro must be applied. The data can then be accessed with the resulting 32-bit far pointer.

Returns: The DefineDLLEntry function returns zero if successful, and a non-zero value otherwise.

Example:
```
#include <windows.h>
int FAR PASCAL FooMe( WORD w1, DWORD w2, WORD w3 )
{
  char str[128];

  sprintf( str, "w1=%hx, w2=%lx, w3=%hx", w1, w2, w3 );
  MessageBox( NULL, str, "DLL Test", MB_OK );
  return( w1 + w2 );
}

int PASCAL WinMain( HANDLE hInstance, HANDLE x1,
        LPSTR lpCmdLine, int x2 )
{
  DefineDLLEntry( 1, (PROCPTR) FooMe, DLL_WORD,
                  DLL_DWORD, DLL_WORD, DLL_ENDLIST );
  MessageBox( NULL, "32-bit DLL Started", "Test", MB_OK );
  return( 1 );
}
```

Classification: WIN386

Synopsis:
```
#include <windows.h>
int DefineUserProc16( int typ, PROCPTR routine, ... );
```

Description: The DefineUserProc16 function defines the arguments accepted by the user defined callback procedure *routine*. There may be up to 32 user defined callbacks. The parameter *typ* indicates which one of `GETPROC_ USERDEFINED_ 1` through `GETPROC_ USERDEFINED_ 32` is being defined (see `GetProc16`). The callback routine must be declared as FAR PASCAL, or as FAR cdecl. The variable argument list defines the types of parameters that will be received by the user defined callback procedure *routine*. Valid parameter types are:

UDP16_PTR	16-bit far pointer
UDP16_DWORD	32-bits
UDP16_WORD	16-bits
UDP16_CHAR	8-bits
UDP16_CDECL	callback routine will be declared as type `cdecl` rather than as type `PASCAL`. This keyword may be placed anywhere before the `UDP16_ ENDLIST` keyword.
UDP16_ENDLIST	Marks the end of the variable argument list.

Once the DefineUserProc16 function has been used to declare the user callback routine, then `GetProc16` may be used to get a 16-bit function pointer that may be used by Windows.

Returns: The DefineUserProc16 function returns zero if it succeeds; and non-zero if it fails.

See Also: `GetProc16`

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

WORD FAR PASCAL Test( DWORD a, WORD b )
{
  char foo[128];

  sprintf( foo, "a=%lx, b=%hx", a, b );
  MessageBox( NULL, foo, "TEST", MB_ OK );
  return( 0x123 );
}
```

```
int DefineTest( void )
{
  FARPROC cb;

  DefineUserProc16( GETPROC_ USERDEFINED_ 1,
                    (PROCPTR) Test,
                    UDP16_ DWORD,
                    UDP16_ WORD,
                    UDP16_ ENDLIST );

  cb = GetProc16( (PROCPTR) Test, GETPROC_ USERDEFINED_ 1 );

  /*
   * cb may then be used whenever a pointer to the
   * callback is required by 16-bit Windows
   */
}
```

Classification: WIN386

Synopsis:
```
#include <windows.h>
void FreeAlias16( DWORD fp16 );
```

Description: FreeAlias16 frees a 16-bit far pointer alias for a 32-bit near pointer that was allocated with `AllocAlias16`. This is important to do when there is no further use for the pointer since there are a limited number of 16-bit aliases available (due to limited space in the local descriptor table).

Returns: The FreeAlias16 function returns nothing.

See Also: `AllocAlias16`

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

DWORD mcs_16;
/*
 * Send a message to a MDI client to create a window.
 * _16SendMessage is used for this example, since it will
 * not do any pointer conversions automatically.
 */
MDICREATESTRUCT mcs;
mcs.szTitle = (LPSTR) AllocAlias16( "c:\\foo.bar" );
mcs.szClass = (LPSTR) AllocAlias16( "mdichild" );
mcs.hOwner  = hInst;
mcs.x = mcs.cx = (int) CW_USEDEFAULT;
mcs.y = mcs.cy = (int) CW_USEDEFAULT;
mcs.style = 0;

/* tell the MDI Client to create the child */
mcs_16 = AllocAlias16( &mcs );
hwnd = (WORD) _16SendMessage( hwndMDIClient,
                              WM_MDICREATE,
                              0,
                              (LONG) mcs_16 );
FreeAlias16( mcs_16 );
FreeAlias16( (DWORD) mcs.szClass );
FreeAlias16( (DWORD) mcs.szTitle );
```

Classification: WIN386

Synopsis:
```
#include <windows.h>
void FreeHugeAlias16( DWORD fp16, DWORD size );
```

Description: FreeHugeAlias16 frees a 16-bit far pointer alias that was allocated with `AllocHugeAlias16`. The size of the original 32-bit memory object must be specified. It is important to use `FreeHugeAlias16` when there is no further use for the pointer, since there are a limited number of 16-bit aliases available (due to limited space in the local descriptor table).

Returns: The FreeHugeAlias16 function returns nothing.

See Also: `AllocHugeAlias16`, `AllocAlias16`, `FreeAlias16`

Example:
```
#include <windows.h>
#include <malloc.h>
#define SIZE 300000

  DWORD alias;
  void *tmp;

  tmp = malloc( SIZE );
  alias = AllocHugeAlias16( tmp, SIZE );

  /* windows calls using the alias ... */

  FreeHugeAlias16( alias, SIZE );
```

Classification: WIN386

Synopsis:
```
#include <windows.h>
void FreeIndirectFunctionHandle( HINDIR handle );
```

Description: FreeIndirectFunctionHandle frees a handle that was obtained using GetIndirectFunctionHandle. This is important to do when there is no further use for the pointer since there are a limited number of 16-bit aliases available (due to limited space in the local descriptor table).

Returns: The FreeIndirectFunctionHandle function returns nothing.

See Also: _ Call16,GetIndirectFunctionHandle, InvokeIndirectFunction

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

  HANDLE hDrv;
  FARPROC lpfn;

  if( (hDrv = LoadLibrary( "foo.lib" )) < 32 )
      return FALSE;
  if( !(lpfn = GetProcAddress( hDrv, "ExtDeviceMode" )) )
      return FALSE;

#ifdef _ _ WINDOWS_ 386_ _
  hIndir = GetIndirectFunctionHandle(
              lpfn,
              INDIR_ WORD,
              INDIR_ WORD,
              INDIR_ DWORD,
              INDIR_ PTR,
              INDIR_ PTR,
              INDIR_ DWORD,
              INDIR_ DWORD,
              INDIR_ WORD,
              INDIR_ ENDLIST );

  cb = (WORD) InvokeIndirectFunction(
                hIndir,
                hwnd,
                hDrv,
                NULL,
                "POSTSCRIPT PRINTER",
                "LPT1",
                NULL,
                NULL,
                0 );
  FreeIndirectFunctionHandle( hIndir );
```

```
#else
  cb = lpfn( hwnd,
             hDrv,
             NULL,
             "POSTSCRIPT PRINTER",
             "LPT1",
             NULL,
             NULL,
             0 );
#endif
```

Classification: WIN386

Synopsis:
```
#include <windows.h>
HINDIR GetIndirectFunctionHandle( FARPROC prc, ... );
```

Description: The GetIndirectFunctionHandle function gets a handle for a 16-bit procedure that is to be invoked indirectly. The procedure is assumed to have PASCAL calling convention, unless the `INDIR_ CDECL` parameter is used, to indicate that Microsoft C calling convention is to be used. The 16-bit far pointer *prc* is supplied to GetIndirectFunctionHandle, and a list of the type of each parameter (in the order that they will be passed to the 16-bit function). The parameter types are:

INDIR_DWORD	A DWORD will be passed.
INDIR_WORD	A WORD will be passed.
INDIR_CHAR	A char will be passed.
INDIR_PTR	A pointer will be passed. This is only used if pointer conversion from 32-bit to 16-bit is required, otherwise; INDIR_DWORD is specified.
INDIR_CDECL	This option may be included anywhere in the list before the `INDIR_ ENDLIST` keyword. When this is used, the calling convention used to invoke the 16-bit function will be the Microsoft C calling convention.
INDIR_ENDLIST	Marks the end of the parameter list.

There is no substitute for this function when compiling for 16-bit Windows. In order to make the code 16-bit Windows compatible, conditional code (based on the __WINDOWS_386__ macro) should be placed around the GetIndirectFunctionHandle usage (see the example).

This handle is a data structure that was created using the `malloc` function. To free the handle, just use one of the `FreeIndirectFunctionHandle` or `free` functions.

You may find it easier to use `_ Call16` rather than GetIndirectFunctionHandle followed by a call to `InvokeIndirectFunction`.

Returns: The GetIndirectFunctionHandle function returns a handle to the indirect function, or NULL if a handle could not be allocated. This handle is used in conjunction with `InvokeIndirectFunction` to call the 16-bit procedure.

See Also: `_ Call16`,`FreeIndirectFunctionHandle`, `InvokeIndirectFunction`

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

  HANDLE hDrv;
  FARPROC lpfn;

  if( (hDrv = LoadLibrary( "foo.lib" )) < 32 )
      return FALSE;
  if( !(lpfn = GetProcAddress( hDrv, "ExtDeviceMode" )) )
      return FALSE;
```

```
#ifdef _ _ WINDOWS_ 386_ _
  hIndir = GetIndirectFunctionHandle(
              lpfn,
              INDIR_ WORD,
              INDIR_ WORD,
              INDIR_ DWORD,
              INDIR_ PTR,
              INDIR_ PTR,
              INDIR_ DWORD,
              INDIR_ DWORD,
              INDIR_ WORD,
              INDIR_ ENDLIST );

  cb = (WORD) InvokeIndirectFunction(
                hIndir,
                hwnd,
                hDrv,
                NULL,
                "POSTSCRIPT PRINTER",
                "LPT1",
                NULL,
                NULL,
                0 );
  FreeIndirectFunctionHandle( hIndir );

#else
  cb = lpfn( hwnd,
             hDrv,
             NULL,
             "POSTSCRIPT PRINTER",
             "LPT1",
             NULL,
             NULL,
             0 );
#endif
```

Classification: WIN386

Synopsis:
```
#include <windows.h>
CALLBACKPTR GetProc16( PROCPTR fcn, long type );
```

Description: The GetProc16 function returns a 16-bit far function pointer suitable for use as a Windows callback function. This callback function will invoke the 32-bit far procedure specified by *fcn*. The types of callback functions that may be allocated are:

GETPROC_CALLBACK This is the most common form of callback; suitable as the callback routine for a window. The callback has the form:

```
long FAR PASCAL WProc( HWND, unsigned,
            WORD, LONG );
```

GETPROC_ABORTPROC This is the callback type used for trapping abort requests when printing. The callback has the form:

```
int FAR PASCAL AbortProc( HDC, WORD );
```

GETPROC_ENUMCHILDWINDOWS This callback is used with the `EnumChildWindows` Windows function. The callback function has the form

```
BOOL FAR PASCAL EnumChildWindowsFunc(
            HWND, DWORD );
```

GETPROC_ENUMFONTS This callback type is used with the `EnumFonts` Windows function. The callback has the form:

```
int FAR PASCAL EnumFontsFunc( LPLOGFONT,
            LPTEXTMETRICS, short, LPSTR );
```

GETPROC_ENUMMETAFILE This callback is used with the `EnumMetaFile` Windows function. The callback function has the form:

```
int FAR PASCAL EnumMetaFileFunc( HDC,
            LPHANDLETABLE, LPMETARECORD,
            short, LPSTR );
```

GETPROC_ENUMOBJECTS This callback is used with the `EnumObjects` Windows function. The callback function has the form:

```
int FAR PASCAL EnumObjectsFunc( LPSTR, LPSTR );
```

GETPROC_ENUMPROPS_FIXED_DS This callback is used with the `EnumProps` Windows function, when the fixed data segments callback is needed. The callback function has the form:

```
int FAR PASCAL EnumPropsFunc(
            HWND, LPSTR, HANDLE );
```

GETPROC_ENUMPROPS_MOVEABLE_DS This callback is used with the `EnumProps` Windows function, when the moveable data segments callback is needed. The callback function has the form:

```
int FAR PASCAL EnumPropsFunc(
            HWND, WORD, PSTR, HANDLE );
```

GETPROC_ENUMTASKWINDOWS This callback is used with the `EnumTaskWindows` Windows function. The callback function has the form:

```
int FAR PASCAL EnumTaskWindowsFunc(
            HWND, DWORD );
```

GETPROC_ENUMWINDOWS This callback is used with the `EnumWindows` Windows function. The callback function has the form:

```
int FAR PASCAL EnumWindowsFunc( HWND, DWORD );
```

GETPROC_GLOBALNOTIFY This callback is used with the `GlobalNotify` Windows function. The callback function has the form:

```
int FAR PASCAL GlobalNotifyFunc( HANDLE );
```

GETPROC_GRAYSTRING This callback is used with the `GrayString` Windows function. The callback function has the form:

```
int FAR PASCAL GrayStringFunc(
            HDC, DWORD, short );
```

GETPROC_LINEDDA This callback is used with the `LineDDA` Windows function. The callback function has the form:

```
void FAR PASCAL LineDDAFunc(
            short, short, LPSTR );
```

GETPROC_SETRESOURCEHANDLER This callback is used with the `SetResourceHandler` Windows function. The callback function has the form:

```
int FAR PASCAL SetResourceHandlerFunc(
            HANDLE, HANDLE, HANDLE );
```

GETPROC_SETTIMER This callback is used with the `SetTimer` Windows function. The callback function has the form:

```
int FAR PASCAL SetTimerFunc(
            HWND, WORD, short, DWORD );
```

GETPROC_SETWINDOWSHOOK This callback is used with the `SetWindowsHook` Windows function. The callback function has the form:

```
int FAR PASCAL SetWindowsHookFunc(
            short, WORD, DWORD );
```

GETPROC_USERDEFINED_x This callback is used in conjunction with `DefineUserProc16` function to create a callback routine with an arbitrary set of parameters. Up to 32 user defined callbacks are allowed, they are identified by using GETPROC_USERDEFINED_1 through GETPROC_USERDEFINED_32. The user defined callback must be declared as a FAR PASCAL function, or as a FAR cdecl function.

Returns: The GetProc16 function returns a 16-bit far pointer to a callback procedure. This pointer may then be fed to any Windows function that requires a pointer to a function within the 32-bit program. Note that the callback function within the 32-bit program must be declared as `FAR`.

See Also: ReleaseProc16

Example:

```
#include <windows.h>

  CALLBACKPTR cbp;
  FARPROC lpProcAbout;
  /*
   * Get a 16-bit callback routine to point at
   * our About dialogue procedure, then create
   * the dialogue.  We use _16 versions of
   * MakeProcInstance, DialogBox, and
   * FreeProcInstance because they do not do
   * any magic work on the callback routines.
   */
  cbp = GetProc16( (PROCPTR) About,
                   GETPROC_CALLBACK );

  lpProcAbout = _16MakeProcInstance( cbp, hInst );

  _16DialogBox( hInst,
              "AboutBox",
              hWnd,
              lpProcAbout );

  _16FreeProcInstance( lpProcAbout );
  ReleaseProc16( cbp );
```

Classification: WIN386

Synopsis:
```
#include <windows.h>
long InvokeIndirectFunction( HINDIR handle, ... );
```

Description: The InvokeIndirectFunction function invokes the 16-bit function pointed to by the specified handle. The handle must have been previously allocated using the `GetIndirectFunctionHandle` function. The handle is followed by the list of parameters to be passed to the 16-bit function.

If you specified `INDIR_ PTR` as a parameter when allocating the handle, then a 16-bit pointer is allocated for a 32-bit pointer that you pass. However, this pointer is freed when the 16-bit function being invoked returns.

There is no substitute for this function when compiling for 16-bit Windows. In order to make the code 16-bit Windows compatible, conditional code (based on the __WINDOWS_386__ macro) should be placed around the InvokeIndirectFunction usage (see the example).

Returns: The InvokeIndirectFunction function returns the value which the 16-bit function returned. If the 16-bit function returns a short rather than a long, the result must be typecast.

See Also: `_ Call16`,`FreeIndirectFunctionHandle`, `GetIndirectFunctionHandle`

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

  HANDLE hDrv;
  FARPROC lpfn;
  HINDIR hIndir;
  int cb;

  if( (hDrv = LoadLibrary( "foo.lib" )) < 32 )
      return FALSE;
  if( !(lpfn = GetProcAddress( hDrv, "ExtDeviceMode" )) )
      return FALSE;
#ifdef _ _ WINDOWS_ 386_ _

  hIndir = GetIndirectFunctionHandle(
              lpfn,
              INDIR_ WORD,
              INDIR_ WORD,
              INDIR_ DWORD,
              INDIR_ PTR,
              INDIR_ PTR,
              INDIR_ DWORD,
              INDIR_ DWORD,
              INDIR_ WORD,
              INDIR_ ENDLIST );
```

```
  cb = (WORD) InvokeIndirectFunction(
                hIndir,
                hwnd,
                hDrv,
                NULL,
                "POSTSCRIPT PRINTER",
                "LPT1",
                NULL,
                NULL,
                0 );
  FreeIndirectFunctionHandle( hIndir );
#else

  cb = lpfn( hwnd,
             hDrv,
             NULL,
             "POSTSCRIPT PRINTER",
             "LPT1",
             NULL,
             NULL,
             0 );
#endif
```

Classification: WIN386

Synopsis:
```
#include <windows.h>
void *MapAliasToFlat( DWORD alias );
```

Description: The MapAliasToFlat function returns a 32-bit near pointer equivalent of a pointer allocated previously with `AllocAlias16` or `AllocHugeAlias16`. This is useful if you are communicating with a 16-bit application that is returning pointers that you previously gave it.

Returns: The MapAliasToFlat function returns a 32-bit near pointer usable by the 32-bit application.

See Also: `AllocAlias16`, `AllocHugeAlias16`

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

  DWORD alias;
  void  *ptr;

  alias = (DWORD) AllocAlias16( &alias );
  alias += 5;
  ptr = MapAliasToFlat( alias );
  if( ptr == ((char *)&alias + 5) ) {
     MessageBox( NULL,"It Worked","",MB_ OK );
  } else {
     MessageBox( NULL,"It Failed","",MB_ OK );
  }
```

Classification: WIN386

Synopsis:
```
#include <windows.h>
DWORD MK_FP16( void far * fp32 );
```

Description: The MK_FP16 function converts a 32-bit far pointer to a 16-bit far pointer. The 16-bit pointer is created by simply removing the high word of the offset of the 32-bit pointer.

The 32-bit far pointer must be one that was obtained by using `MK_FP32` to extend a 16-bit pointer.

This is useful whenever it is necessary to pass a 16-bit far pointer a parameter to a Windows function though an _16 function.

Returns: The MK_FP16 returns a 16-bit far pointer.

See Also: `MK_LOCAL32`,`MK_FP32`

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

  DRAWITEMSTRUCT    FAR *lpdis;
  RECT  rc;
  DWORD alias;
  /*
   * The drawitem struct was passed as a long, so we
   * have to convert it to a 32 bit far pointer.
   * Then, we want the 16 bit far pointer of the rcItem
   * element so we can pass it to CopyRect (_16CopyRect
   * is a version of CopyRect that does not convert
   * the pointers it was given).
   */
  case WM_DRAWITEM:
    lpdis = MK_FP32( (void *) lParam );
    alias = AllocAlias16( > );
    _16CopyRect( (LPRECT) alias,
                 (LPRECT) MK_FP16( &lpdis->rcItem ) );
    FreeAlias16( alias );
```

Classification: WIN386

Synopsis:
```
#include <windows.h>
void far *MK_FP32( void * fp16 );
```

Description: The MK_FP32 function converts a 16-bit far pointer to a 32-bit far pointer. This is needed whenever Windows returns a 16-bit far pointer, and access to the data is needed by the 32-bit program.

Returns: The MK_FP32 returns a 32-bit far pointer.

See Also: MK_LOCAL32,MK_FP16

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

  MEASUREITEMSTRUCT far *mis;

  case WM_MEASUREITEM:
    /*
     * Windows has passed us a 16 bit far pointer
     * to the measure item data structure.  We
     * use MK_FP32 to make that pointer a 32-bit far
     * pointer, which enables us to access the data.
     */
    mis = MK_FP32( (void *) lParam );
    mis->itemHeight = MEASUREITEMHEIGHT;
    mis->itemWidth  = MEASUREITEMWIDTH;
    return TRUE;
```

Classification: WIN386

Synopsis:
```
#include <windows.h>
void far *MK_ LOCAL32( void * fp16 );
```

Description: The MK_LOCAL32 function converts a 16-bit near pointer to a 32-bit far pointer. This is needed whenever Windows returns a 16-bit near pointer that is to be accessed by the 32-bit program.

Returns: The MK_LOCAL32 returns a 32-bit far pointer.

See Also: MK_ FP32,MK_ FP16

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

  WORD ich,cch;
  char *pch;
  char far *fpch;
  HANDLE hT;

  /*
   * Request the data from an edit window; copy it
   * into a local buffer so that it can be passed
   * to TextOut
   */
  ich = (WORD) SendMessage( hwndEdit,
                            EM_ LINEINDEX,
                            iLine,
                            0L );
  cch = (WORD) SendMessage( hwndEdit,
                            EM_ LINELENGTH,
                            ich,
                            0L );
  fpch = MK_ LOCAL32( LocalLock( hT ) ) ;
  pch = alloca( cch );
  _ fmemcpy( pch, fpch + ich, cch );

  TextOut( hdc, 0, yExtSoFar, (LPSTR) pch, cch );
  LocalUnlock( hT );
```

Classification: WIN386

Synopsis:
```
#include <windows.h>
void *PASS_ WORD_ AS_ POINTER( DWORD dw );
```

Description: Some Windows API functions have pointer parameters that do not always take pointers. Sometimes these parameters are pure data. In order to stop the supervisor from trying to convert the data into a 16-bit far pointer, the PASS_WORD_AS_POINTER function is used.

Returns: The PASS_WORD_AS_POINTER returns a 32-bit "near" pointer, that is really the parameter *dw*.

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

  Func( PASS_ WORD_ AS_ POINTER( 1 ) );
```

Classification: WIN386

Synopsis:
```
#include <windows.h>
void ReleaseProc16( CALLBACKPTR cbp );
```

Description: ReleaseProc16 releases the callback function allocated by `GetProc16`. Since the callback routines are a limited resource, it is important to release the routines when they are no longer required.

Returns: The ReleaseProc16 function returns nothing.

See Also: `GetProc16`

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

  CALLBACKPTR cbp;
  FARPROC lpProcAbout;
  /*
   * Get a 16-bit callback routine to point at
   * our About dialogue procedure, then create
   * the dialogue.  We use _16 versions of
   * MakeProcInstance, DialogBox, and
   * FreeProcInstance because they do not do
   * any magic work on the callback routines.
   */
   cbp = GetProc16( (PROCPTR) About,
                    GETPROC_CALLBACK );

   lpProcAbout = _16MakeProcInstance( cbp, hInst );

   _16DialogBox( hInst,
              "AboutBox",
              hWnd,
              lpProcAbout );

   _16FreeProcInstance( lpProcAbout );
   ReleaseProc16( cbp );
```

Classification: WIN386

24 32-bit Extended Windows Application Development

The purpose of this chapter is to anticipate some common questions about 32-bit Windows application development.

The following topics are discussed in this chapter:

- Can you call 16-bit code from a 32-bit code?
- Can I WinExec another Windows application?
- How do I add my Windows resources?
- What size of function pointers passed to Windows?
- Why are 32-bit callback routines FAR?
- Why use the _16 API functions?
- What about pointers in structures?
- When do I use MK_FP32?
- What is the difference between AllocAlias16 and MK_FP16?

24.1 Can you call 16-bit code from a 32-bit code?

A 32-bit Windows application can make a call to 16-bit code through the use of the Open Watcom `_Call16` or`InvokeIndirectFunction` procedures. These functions ensure that the Open Watcom Windows Supervisor prepares the stack for the 16-bit call and return to the 32-bit code. The 32-bit application uses `LoadLibrary` function to bring the 16-bit DLL into memory and then calls the 16-bit procedures. To invoke 16-bit procedures, use `GetProcAddress` to get the 16-bit far pointer to the function. Use the `_Call16` procedure to call the 16-bit function since it is simpler to use than the `GetIndirectFunctionHandle`, `InvokeIndirectFunction`, and `FreeIndirectFunctionHandle` sequence. An example of this process is provided under the `_Call16` Windows library function description.

This method can be used to call any 16-bit Dynamic Link Library (DLL) procedure or any 32-bit extended DLL procedure from within a 32-bit application, including DLLs that are available as products through Independent Software Vendors (ISVs).

24.2 Can I WinExec another Windows application?

As far as Windows is concerned, the WinExec was made by a 16-bit application, and the application specified will be started. This new application can be a 16-bit application or another 32-bit application that was implemented with Open Watcom C/C++

24.3 How do I add my Windows resources?

The `WBIND` utility automatically runs the resource compiler to add the resources to the 32-bit Windows supervisor (since the supervisor is a 16-bit Windows application). Note that resource compiler options may be specified by using the "R" option of `WBIND`.

24.4 What size of function pointers passed to Windows?

All function pointers passed to Windows must be 16-bit far pointers since no translation is applied to any function pointers passed to Windows. Translation is often not possible, since any functions that Windows is to call back must be exported, and only 16-bit functions can be exported.

A 16-bit far pointer to a function is obtained in one of two ways: either Windows gives it to you (via `GetProcAddr`, for example), or you obtain a pointer from the supervisor, via `GetProc16`.

Function pointers obtained from Windows may either be fed into other Windows functions requiring function pointers, or called indirectly by using `_Call16` or by using the `GetIndirectFunctionHandle`, `InvokeIndirectFunction`, and `FreeIndirectFunctionHandle` sequence.

The function `GetProc16` returns a 16-bit far pointer to a callback function that Windows can use. This callback function will direct control into the desired 32-bit routine.

24.5 Why are 32-bit callback routines FAR?

The callback routines are declared as FAR so that the compiler will generate a far return from the procedure. This is necessary since the 32-bit callback routine is "far" called from the supervisor.

The callback routine is still "near" in the sense that it lies within the 32-bit flat address space of the application. This means that `GetProc16` only needs the offset of the 32-bit callback function in order to set up the 16-bit procedure to call back correctly. Thus, `GetProc16` accepts type `PROCPTR` which is in fact only 4 bytes long. The compiler will provide the offset only, which is, as already stated, all that is needed.

24.6 Why use the _16 API functions?

The regular Windows API functions used in Open Watcom C/C++ automatically convert any pointers to 16-bit far pointers for use by Windows. Sometimes, you may have a set of pointers that are 16-bit far pointers already (e.g., obtained from `GlobalLock`), and do not need any conversion. The "_16..." API functions do not convert pointers, they simply pass them on directly to Windows. See the appendix entitled "Special Windows API Functions" on page 205 for a list of the "_16..." API functions.

24.7 What about pointers in structures?

Pointers in structures will be converted if the Windows API function actually takes a pointer to that structure (i.e., if it is possible for the supervisor to identify that structure). There are few functions that accept pointers to structures containing pointers. One such function is *RegisterClass* which accepts a pointer to a WNDCLASS structure.

If Windows has you passing a pointer to a structure through a 32-bit integer argument, then it is not possible for the supervisor to identify that as a pointer that needs conversion. It is also not possible for the supervisor to convert any pointers contained in the structure, since it is not aware that it is a structure (as far as the supervisor is concerned, that data is what Windows said it was - a 32-bit integer). In this case, it is necessary to get 16-bit far pointer equivalents to the 32-bit near pointers that you want to pass. Use `AllocAlias16` for this.

24.8 When do I use MK_FP32?

`MK_ FP32` is used to convert all 16-bit far pointers to 32-bit far pointers that can be used by your 32-bit application. For example, to access the memory returned by `GlobalLock` requires the use of `MK_ FP32`. To access any pointer passed to you (in a callback routine) requires the use of `MK_ FP32` if you want access to that data in your 32-bit application.

24.9 What is the difference between AllocAlias16 and MK_FP16?

`AllocAlias16` actually gets a new selector that points at the same memory as the 32-bit near pointer, whereas `MK_ FP16` squishes a 32-bit far pointer back into a 16-bit far pointer (i.e., it reverses `MK_ FP32`).

24.10 Tell Me More About Thunking and Aliases

Consider the following example.

```
dwAlias = AllocAlias16( pszSomething );
hwnd = CreateWindowEx(
            0L,                     // extendedStyle
            "classname",            // class name
            "",
            WS_ POPUP|WS_ VISIBLE|WS_ CLIPSIBLINGS|WS_ HSCROLL|
            WS_ BORDER|WS_ CAPTION|WS_ SYSMENU,
            x, y, 0, 0,             // x, y, cx, cy
            hwndParent,             // hwndParent
            NULL,                   // control ID
            g_ app.hinst,           // hInstance
            (void FAR*)dwAlias);    // lpCreateParams

FreeAlias16( dwAlias );
```

When I get the `lpCreateParams` parameter in `WM_ CREATE`, I don't get the original`dwAlias` but something else which looks like another alias to me. So the question is: Must the `CreateWindowEx` parameter `lpCreateParams` be "thunked" or is this done automatically by the supervisor?

Thunks are always created for function pointers. Aliases are always created for data pointers. There are 3 data pointer parameters in the `CreateWindowEx` function call. Aliases are created for all three pointers. The `lpCreateParams` argument is a pointer to a struct which contains 3 pointers. Aliases are not created for the 3 pointers inside the struct. If you need to have this done, then you will have to create the aliases yourself. If you create aliases for the parameters to `CreateWindowEx,` then you must call the `_ 16CreateWindowEx` function which will not create any aliases.

Here is some further information on thunks (which are created for function pointers). There is code in the supervisor that *trys* (note the word *trys*) to determine if the user has already created a thunk and, if so, avoids creating a double thunk which will always generate a GPF. The best policy is to let the supervisor automatically create all thunks for you unless you have a very specific reason not to, in which case you should call the _16 version of the function.

Here is some further information on aliases (which are created for data pointers). There is no way for the supervisor to determine if a value is a 32-bit flat pointer or an alias for the pointer. So if you pass in an alias to the non _16 version of the function, the supervisor will create an alias for the alias which will end up pointing to the wrong memory location. If you are going to create the alias, then you must call the _16 version of the function.

25 Special Variables for Windows Programming

__A000	A selector for addressing the real-mode segment 0xA000.
__B000	A selector for addressing the real-mode segment 0xB000.
__B800	A selector for addressing the real-mode segment 0xB800.
__C000	A selector for addressing the real-mode segment 0xC000.
__D000	A selector for addressing the real-mode segment 0xD000.
__E000	A selector for addressing the real-mode segment 0xE000.
__F000	A selector for addressing the real-mode segment 0xF000.
LocalPtr	The selector for the supervisor's data area.

26 Definitions of Windows Terms

CALLBACKPTR	Pointer to a 16-bit callback routine; used to call into 32-bit functions.
DWORD	An unsigned long.
HINDIR	A handle to 16-bit function that needs to be called indirectly.
PROCPTR	A pointer to a 32-bit callback routine. Although the callback routine is declared as far, only the 32-bit offset is used.
WORD	An unsigned short.

27 Special Windows API Functions

On rare occasions, you want to use 16-bit far pointers directly in a Windows function. Since all Windows functions in the 32-bit environment are expecting 32-bit near pointers, you cannot simply use the 16-bit far pointer directly in the function.

The following functions are special versions of Windows API functions that do NOT convert any of the pointers from 32-bit to 16-bit. There are _16 versions of all Windows API functions that accept data pointers.

_16AddAtom
_16AddFontResource
_16AdjustWindowRect
_16AdjustWindowRectEx
_16AnimatePalette
_16AnsiLower
_16AnsiLowerBuff
_16AnsiToOem
_16AnsiToOemBuff
_16AnsiUpper
_16AnsiUpperBuff
_16BuildCommDCB
_16CallMsgFilter
_16ChangeMenu
_16ClientToScreen
_16ClipCursor
_16CopyMetaFile
_16CopyRect
_16CreateBitmap
_16CreateBitmapIndirect
_16CreateBrushIndirect
_16CreateCursor
_16CreateDC
_16CreateDialog
_16CreateDialogIndirect
_16CreateDialogIndirectParam
_16CreateDialogParam
_16CreateDIBitmap
_16CreateEllipticRgnIndirect
_16CreateFont
_16CreateFontIndirect
_16CreateIC
_16CreateIcon
_16CreateMetaFile
_16CreatePalette
_16CreatePenIndirect
_16CreatePolygonRgn
_16CreatePolyPolygonRgn
_16CreateRectRgnIndirect

_16CreateWindow
_16CreateWindowEx
_16DialogBox
_16DialogBoxIndirect
_16DialogBoxIndirectParam
_16DialogBoxParam
_16DispatchMessage
_16DlgDirList
_16DlgDirListComboBox
_16DlgDirSelect
_16DlgDirSelectComboBox
_16DPtoLP
_16DrawFocusRect
_16DrawText
_16EndPaint
_16EnumChildWindows
_16EnumFonts
_16EnumMetaFile
_16EnumObjects
_16EnumProps
_16EnumTaskWindows
_16EnumWindows
_16EqualRect
_16Escape
_16ExtTextOut
_16FillRect
_16FindAtom
_16FindResource
_16FindWindow
_16FrameRect
_16FreeProcInstance
_16GetAtomName
_16GetBitmapBits
_16GetCaretPos
_16GetCharWidth
_16GetClassInfo
_16GetClassName
_16GetClientRect
_16GetClipboardFormatName
_16GetClipBox
_16GetCodeInfo
_16GetCommError
_16GetCommState
_16GetCursorPos
_16GetDIBits
_16GetDlgItemInt
_16GetDlgItemText
_16GetEnvironment
_16GetKeyboardState
_16GetKeyNameText
_16GetMenuString
_16GetMetaFile
_16GetModuleFileName
_16GetModuleHandle

_16GetObject
_16GetPaletteEntries
_16GetPriorityClipboardFormat
_16GetPrivateProfileInt
_16GetPrivateProfileString
_16GetProcAddress
_16GetProfileInt
_16GetProfileString
_16GetProp
_16GetRgnBox
_16GetScrollRange
_16GetSystemDirectory
_16GetSystemPaletteEntries
_16GetTabbedTextExtent
_16GetTempFileName
_16GetTextExtent
_16GetTextFace
_16GetTextMetrics
_16GetUpdateRect
_16GetWindowRect
_16GetWindowsDirectory
_16GetWindowText
_16GlobalAddAtom
_16GlobalFindAtom
_16GlobalGetAtomName
_16GlobalNotify
_16GrayString
_16InflateRect
_16IntersectRect
_16InvalidateRect
_16InvertRect
_16IsDialogMessage
_16IsRectEmpty
_16LineDDA
_16LoadAccelerators
_16LoadBitmap
_16LoadCursor
_16LoadIcon
_16LoadLibrary
_16LoadMenu
_16LoadMenuIndirect
_16LoadModule
_16LoadString
_16LPtoDP
_16MakeProcInstance
_16MapDialogRect
_16MessageBox
_16OemToAnsi
_16OemToAnsiBuff
_16OffsetRect
_16OpenComm
_16OpenFile
_16OutputDebugString
_16PlayMetaFileRecord

_16Polygon
_16Polyline
_16PolyPolygon
_16PtInRect
_16ReadComm
_16RectInRegion
_16RectVisible
_16RegisterClipboardFormat
_16RegisterWindowMessage
_16RemoveFontResource
_16RemoveProp
_16ScreenToClient
_16ScrollDC
_16ScrollWindow
_16SetBitmapBits
_16SetCommState
_16SetDIBits
_16SetDIBitsToDevice
_16SetDlgItemText
_16SetEnvironment
_16SetKeyboardState
_16SetPaletteEntries
_16SetProp
_16SetRect
_16SetRectEmpty
_16SetResourceHandler
_16SetSysColors
_16SetTimer
_16SetWindowsHook
_16SetWindowText
_16StretchDIBits
_16TabbedTextOut
_16TextOut
_16ToAscii
_16TrackPopupMenu
_16TranslateAccelerator
_16TranslateMDISysAccel
_16TranslateMessage
_16UnhookWindowsHook
_16UnionRect
_16UnregisterClass
_16ValidateRect
_16WinExec
_16WinHelp
_16WriteComm
_16WritePrivateProfileString
_16WriteProfileString
_16_lread
_16_lwrite

Windows NT Programming Guide

28 Windows NT Programming Overview

Windows NT supports both non-windowed character-mode applications and windowed Graphical User Interface (GUI) applications. In addition, Windows NT supports Dynamic Link Libraries and applications with multiple threads of execution.

We have supplied all the necessary tools for native development on Windows NT. You can also cross develop for Windows NT using either the DOS-hosted compilers and tools, the Windows 95-hosted compilers and tools, or the OS/2-hosted compilers and tools.

Note - If you are on the host with operating system other then 32-bit Windows, you should setup INCLUDE environment variable correctly to compile for 32-bit Windows target.

You can do that by command (DOS, OS/2, Windows 3.x)

set INCLUDE=%WATCOM%\h;%WATCOM%\h\nt

or by command (LINUX)

export INCLUDE=$WATCOM/h:$WATCOM/h/nt

Testing and debugging of your Windows NT application must be done on Windows NT or Windows 95.

If you are creating a character-mode application, you may also be interested in a special DOS extender from Phar Lap (TNT) that can run your Windows NT character-mode application under DOS.

28.1 Windows NT Programming Note

When doing Win32 programming, you should use the /ei and /zp4 options to compile C and C++ code with the Open Watcom compilers since this adjusts the compilers to match the default Microsoft compiler behaviour. Some Microsoft software relies on the default behaviour of their own compiler regarding the treatment of enums and structure packing alignment.

28.2 Windows NT Character-mode Versus GUI

Basically, there are two classes of C/C++ applications that can run in a windowed environment like Windows NT.

The first are those C/C++ applications that do not use any of the Win32 API functions; they are strictly C/C++ applications that do not rely on the features of a particular operating system.

- This Application must be created as Windows NT Character-mode Application.

The second class of C/C++ applications are those that actually call Win32 API functions directly. These are applications that have been tailored for the Win32 operating environment. There can occure two application types.

- First one uses GUI interface then it must be created as Windows NT GUI Application.
- Second one uses only character console (no GUI) then it must be created as Windows NT Character-mode Application

A subsequent chapters deal with the creation of different application types for Windows NT target.

29 Creating Windows NT GUI Applications

This chapter describes how to compile and link Windows NT GUI applications simply and quickly. In this chapter, we look at applications written to exploit the Windows NT Application Programming Interface (API).

We will illustrate the steps to creating Windows NT GUI applications by taking a small sample application and showing you how to compile, link, run and debug it.

29.1 The Sample GUI Application

To demonstrate the creation of Windows NT GUI applications, we introduce a simple sample program. The following example is the "hello" program adapted for Windows.

```
#include <windows.h>

int PASCAL WinMain( HANDLE hInstance, HANDLE hPrevInst,
                    LPSTR lpCmdLine, int nCmdShow )
{
    MessageBox( NULL, "Hello world",
                "Open Watcom C/C++ for Windows",
                MB_ OK | MB_ TASKMODAL );
    return( 0 );
}
```

The goal of this program is to display the message "Hello world" on the screen. The `MessageBox` Windows API function is used to accomplish this task. We will take you through the steps necessary to produce this result.

29.2 Building and Running the GUI Application

To compile and link our example program which is stored in the file `hello.c`, enter the following command:

```
C>wcl386 -l=nt_ win -bt=nt  hello.c
```

The typical messages that appear on the screen are shown in the following illustration.

```
C>wcl386 -l=nt_win -bt=nt  hello.c
Open Watcom C/C++32 Compile and Link Utility
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1988-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
        wcc386 hello.c  -bt=nt
Open Watcom C32 Optimizing Compiler
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1984-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
hello.c: 10 lines, included 6500, 0 warnings, 0 errors
Code size: 41

Open Watcom Linker
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1985-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
loading object files
searching libraries
creating a Windows NT windowed executable
```

If you examine the current directory, you will find that two files have been created. These are `hello.obj` (the result of compiling `hello.c`) and `hello.exe` (the result of linking `hello.obj` with the appropriate Open Watcom C/C++ libraries).

The resultant Windows NT GUI application `HELLO.EXE` can now be run under Windows NT.

29.3 Debugging the GUI Application

Let us assume that you wish to debug your application in order to locate an error in programming. In the previous section, the "hello" program was compiled with default compile and link options. When debugging an application, it is useful to refer to the symbolic names of routines and variables. It is also convenient to debug at the source line level rather than the machine language level. To do this, we must direct both the compiler and linker to include additional debugging information in the object and executable files. Using the `WCL386` command, this is fairly straightforward. `WCL386` recognizes the Open Watcom C/C++ compiler "debug" options and will create the appropriate debug directives for the Open Watcom Linker.

For example, to compile and link the "hello" program with debugging information, the following command may be issued.

```
C>wcl386 -l=nt_win -bt=nt  -d2 hello.c
```

The typical messages that appear on the screen are shown in the following illustration.

```
C>wcl386 -l=nt_win -bt=nt  -d2 hello.c
Open Watcom C/C++32 Compile and Link Utility
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1988-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
        wcc386 hello.c -bt=nt  -d2
Open Watcom C32 Optimizing Compiler
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1984-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
hello.c: 10 lines, included 6500, 0 warnings, 0 errors
Code size: 66

Open Watcom Linker
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1985-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
loading object files
searching libraries
creating a Windows NT windowed executable
```

The "d2" option requests the maximum amount of debugging information that can be provided by the Open Watcom C/C++ compiler. `WCL386` will make sure that this debugging information is included in the executable file that is produced by the linker.

The "Code size" value is larger than in the previous example since selection of the "d2" option results in fewer code optimizations by default. You can request more optimization by specifying the appropriate options. However, you do so at the risk of making it more difficult for yourself to determine the relationship between the object code and the original source language code.

To request the Open Watcom Debugger to assist in debugging the application, select the Open Watcom Debugger icon. It would be too ambitious to describe the debugger in this introductory chapter so we refer you to the book entitled ***Open Watcom Debugger User's Guide.***

30 Creating Windows NT Character-mode Applications

This chapter describes how to compile and link Windows NT Character-mode applications simply and quickly. In this chapter, we look at applications written to exploit the Windows NT Application Programming Interface (API).

We will illustrate the steps to creating Windows NT Character-mode applications by taking a small sample application and showing you how to compile, link, run and debug it.

30.1 The Sample Character-mode Application

To demonstrate the creation of Windows NT Character-mode applications, we introduce a simple sample program. The following example is the "hello" program adapted for Windows.

```
#include <stdio.h>

void main()
{
    printf( "Hello world\n" );
}
```

The goal of this program is to display the message "Hello world" on the screen. The C library `printf` routine is used to accomplish this task. We will take you through the steps necessary to produce this result.

30.2 Building and Running the Character-mode Application

To compile and link our example program which is stored in the file `hello.c`, enter the following command:

```
C>wcl386 -l=nt -bt=nt hello.c
```

The typical messages that appear on the screen are shown in the following illustration.

```
C>wcl386 -l=nt -bt=nt hello.c
Open Watcom C/C++32 Compile and Link Utility
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1988-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
        wcc386 hello.c  -bt=nt
Open Watcom C32 Optimizing Compiler
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1984-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
hello.c: 10 lines, included 6500, 0 warnings, 0 errors
Code size: 41

Open Watcom Linker
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1985-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
loading object files
searching libraries
creating a Windows NT Character-mode executable
```

If you examine the current directory, you will find that two files have been created. These are `hello.obj` (the result of compiling `hello.c`) and `hello.exe` (the result of linking `hello.obj` with the appropriate Open Watcom C/C++ libraries).

The resultant Windows NT Character-mode application `HELLO.EXE` can now be run under Windows NT.

30.3 Debugging the Character-mode Application

Let us assume that you wish to debug your application in order to locate an error in programming. In the previous section, the "hello" program was compiled with default compile and link options. When debugging an application, it is useful to refer to the symbolic names of routines and variables. It is also convenient to debug at the source line level rather than the machine language level. To do this, we must direct both the compiler and linker to include additional debugging information in the object and executable files. Using the `WCL386` command, this is fairly straightforward. `WCL386` recognizes the Open Watcom C/C++ compiler "debug" options and will create the appropriate debug directives for the Open Watcom Linker.

For example, to compile and link the "hello" program with debugging information, the following command may be issued.

```
C>wcl386 -l=nt -bt=nt -d2 hello.c
```

The typical messages that appear on the screen are shown in the following illustration.

```
C>wcl386 -l=nt -bt=nt -d2 hello.c
Open Watcom C/C++32 Compile and Link Utility
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1988-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
        wcc386 hello.c -bt=nt -d2
Open Watcom C32 Optimizing Compiler
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1984-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
hello.c: 10 lines, included 6500, 0 warnings, 0 errors
Code size: 66

Open Watcom Linker
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1985-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
loading object files
searching libraries
creating a Windows NT Character-mode executable
```

The "d2" option requests the maximum amount of debugging information that can be provided by the Open Watcom C/C++ compiler. `WCL386` will make sure that this debugging information is included in the executable file that is produced by the linker.

The "Code size" value is larger than in the previous example since selection of the "d2" option results in fewer code optimizations by default. You can request more optimization by specifying the appropriate options. However, you do so at the risk of making it more difficult for yourself to determine the relationship between the object code and the original source language code.

To request the Open Watcom Debugger to assist in debugging the application, select the Open Watcom Debugger icon. It would be too ambitious to describe the debugger in this introductory chapter so we refer you to the book entitled ***Open Watcom Debugger User's Guide.***

31 Windows NT Multi-threaded Applications

This chapter describes how to create multi-threaded applications. A multi-threaded application is one whose tasks are divided among several threads of execution. A process is an executing application and the resources it uses. A thread is the smallest unit of execution within a process. Each thread has its own stack and a set of machine registers and shares all resources with its parent process. The path of execution of one thread does not affect that of another; each thread is an independent entity.

Typically, an application has a single thread of execution. In this type of application, all tasks, once initiated, are completed before the next task begins. In contrast, tasks in a multi-threaded application can be performed concurrently since more than one thread is executing at once. For example, each thread may be designed to perform a separate task.

31.1 Programming Considerations

Since a multi-threaded application consists of many threads of execution, there are a number of issues that you must consider.

Since threads share the resources of its parent, it may be necessary to serialize access to these resources. For example, if your application has a function that displays information on the console and is used by all threads, it is necessary to allow only one thread to use that function at any time. That is, once a thread calls that function, the function should ensure that no other thread displays information until all information for the initial thread has been displayed. An example of such a function is the `printf` library function.

Another issue that must be considered when creating multi-threaded applications is global variables. If you have global variables that contain thread-specific information, there must be an instance of each global variable for each thread. An example of such a variable is the `errno` global variable defined in the run-time libraries. If an error condition was created by a thread, you would not want it to affect the execution of other threads. Therefore, each thread should contain its own instance of this variable.

31.2 Creating Threads

Each application initially contains a single thread. The run-time libraries contain two functions that create and terminate threads of execution. The function `_ beginthread` creates a thread of execution and the function `_ endthread` ends a thread of execution. The macro`_ threadid` can be used to determine the current thread identifier.

> ***WARNING!*** If any thread calls a library function, you must use the `_beginthread` function to create the thread. Do not use the `CreateThread` API function.

31.2.1 Creating a New Thread

The `_beginthread` function creates a new thread. It is defined as follows.

```
unsigned long _beginthread( void (*start_address)(void *),
                            unsigned stack_size,
                            void *arglist);
```

where	***description:***
start_address	is the address of the function that will be called when the newly created thread is executed. When the thread returns from that function, the thread will be terminated. Note that a call to the `_endthread` function will also terminate the thread.
stack_size	specifies the size of the stack to be allocated by the operating system for the new thread. The stack size should be a multiple of 4K.
arglist	is passed as an argument to the function specified by `start_address`. If no argument is required, a value of NULL can be specified.

If a new thread is successfully created, the thread identifier of the new thread is returned. Otherwise, a value of -1 is returned.

The header file `process.h` contains the definition of the `_beginthread` function.

Another thread related function for Windows NT is `_beginthreadex`. See the ***Open Watcom C Library Reference*** for more information.

31.2.2 Terminating the Current Thread

The `_endthread` function terminates the current thread. It is defined as follows.

```
void _endthread( void )
```

The header file `process.h` contains the definition of the `_endthread` function.

31.2.3 Getting the Current Thread Identifier

The `_threadid` macro can be used to determine the current thread identifier. It is defined as follows.

```
int *__threadid(void);
#define _threadid (__threadid())
```

The header file `stddef.h` contains the definition of the `_threadid` macro.

31.3 A Multi-threaded Example

Let us create a simple multi-threaded application.

```
#include <process.h>
#include <stdio.h>
#include <stddef.h>
#include <windows.h>

static  volatile int    NumThreads;
static  volatile int    HoldThreads;

CRITICAL_ SECTION CriticalSection;

#define NUM_ THREADS     5
#define STACK_ SIZE      8192

static void a_ thread( void *arglist )
/*************************************/
{
    while( HoldThreads ) {
        Sleep( 1 );
    }
    printf( "Hi from thread %d\n", *_ threadid );
    EnterCriticalSection( &CriticalSection );
    --NumThreads;
    LeaveCriticalSection( &CriticalSection );
    _ endthread();
}

int main( void )
/**************/
{
    int         i;

    printf( "Initial thread id = %d\n", *_ threadid );
    NumThreads = 0;
    HoldThreads = 1;
    InitializeCriticalSection( &CriticalSection );
    /* initial thread counts as 1 */
    for( i = 2; i <= NUM_ THREADS; ++i ) {
        if( _ beginthread( a_ thread, STACK_ SIZE, NULL ) == -1 ) {
            printf( "creation of thread %d failed\n", i );
        } else {
            ++NumThreads;
        }
    }
    HoldThreads = 0;
    while( NumThreads != 0 ) {
        Sleep( 1 );
    }
    DeleteCriticalSection( &CriticalSection );
    return( 0 );
}
```

Note:

1. In the function `a_ thread`,`EnterCriticalSection` and `LeaveCriticalSection` are called when we modify the variable `NumThreads`. This ensures that the action of extracting the value of `NumThreads` from memory, incrementing the value, and storing the new result into memory, occurs without interruption. If these functions were not called, it would be possible for two threads to extract the value of `NumThreads` from memory before an update occurred.

Let us assume that the file `mthread.c` contains the above example. Before compiling the file, make sure that the **WATCOM** environment variable is set to the directory in which you installed Open Watcom

C/C++. Also, the **INCLUDE** environment variable must include the `\watcom\h\nt` and `\watcom\h` directories ("\WATCOM" is the directory in which Open Watcom C/C++ was installed).

We can now compile and link the application by issuing the following command.

```
C:\>wcl386 -bt=nt -bm -l=nt mthread
```

The "bm" option must be specified since we are creating a multi-threaded application. If your multi-threaded application contains more than one module, each module must be compiled using the "bm" switch.

The "l" option specifies the target system for which the application is to be linked. The system name `nt` is defined in the file `wlsystem.lnk` which is located in the "BINW" subdirectory of the directory in which you installed Open Watcom C/C++.

The multi-threaded application is now ready to be run.

32 Windows NT Dynamic Link Libraries

A dynamic link library, like a standard library, is a library of functions. When an application uses functions from a standard library, the library functions referenced by the application become part of the executable module. This form of linking is called static linking. When an application uses functions from a dynamic link library, the library functions referenced by the application are not included in the executable module. Instead, the executable module contains references to these functions which are resolved when the application is loaded. This form of linking is called dynamic linking.

Let us consider some of the advantages of using dynamic link libraries over standard libraries.

1. Functions in dynamic link libraries are not linked into your program. Only references to the functions in dynamic link libraries are placed in the program module. These references are called import definitions. As a result, the linking time is reduced and disk space is saved. If many applications reference the same dynamic link library, the saving in disk space can be significant.

2. Since program modules only reference dynamic link libraries and do not contain the actual executable code, a dynamic link library can be updated without re-linking your application. When your application is executed, it will use the updated version of the dynamic link library.

3. Dynamic link libraries also allow sharing of code and data between the applications that use them. If many applications that use the same dynamic link library are executing concurrently, the sharing of code and data segments improves memory utilization.

32.1 Creating Dynamic Link Libraries

Once you have developed the source for a library of functions, a number of steps are required to create a dynamic link library containing those functions.

First, you must compile your source using the "bd" compiler option. This option tells the compiler that the module you are compiling is part of a dynamic link library. Once you have successfully compiled your source, you must create a linker directive file that describes the attributes of your dynamic link library. The following lists the most common linker directives required to create a dynamic link library.

1. The "SYSTEM" directive is used to specify that a dynamic link library is to be created.
2. The "EXPORT" directive is used to to specify which functions in the dynamic link library are to be exported.

Specifying exports in the source code

The "EXPORT" directive need not be used when the symbols to be exported are declared with the `__declspec( dllexport )` modifier in the source code. Such symbols are exported automatically, through special records inserted into the object files by the compiler.

Exporting C++ symbols and classes

Symbols exported via the "EXPORT" directive have to be entered in their mangled form. This makes it rather awkward to export C++ functions, classes or global objects. These symbols also often reference other compiler-generated symbols (invisible to the user) that need be exported together with the class/object. Using the `__declspec( dllexport )` method of exporting symbols is the preferred solution.

3. The "OPTION" directive is used to specify attributes such as the name of the dynamic link library and how to allocate the automatic data segment when the dynamic link library is referenced.
4. The "SEGMENT" directive is used to specify attributes of segments. For example, a segment may be read-only or read-write.

Once the dynamic link library is created, you must allow access to the dynamic link library to client applications that wish to use it. This can be done by creating an import library for the dynamic link library or creating a linker directive file that contains "IMPORT" directives for each of the entry points in the dynamic link library.

32.2 Creating a Sample Dynamic Link Library

Let us now create a dynamic link library using the following example.

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

#if defined(__cplusplus)
#define EXPORTED extern "C" __declspec( dllexport )
#else
#define EXPORTED __declspec( dllexport )
#endif

DWORD TlsIndex; /* Global Thread Local Storage index */

/* Error checking should be performed in following code */

BOOL APIENTRY LibMain( HANDLE hinstDLL,
                       DWORD  fdwReason,
                       LPVOID lpvReserved )
{
    switch( fdwReason ) {
    case DLL_PROCESS_ATTACH:
        /* do process initialization */

        /* create TLS index */
        TlsIndex = TlsAlloc();
        break;

    case DLL_THREAD_ATTACH:
        /* do thread initialization */

        /* allocate private storage for thread */
        /* and save pointer to it */
        TlsSetValue( TlsIndex, malloc(200) );
        break;
```

```
        case DLL_ THREAD_ DETACH:
            /* do thread cleanup */

            /* get the TLS value and free associated memory */
            free( TlsGetValue( TlsIndex ) );
            break;

        case DLL_ PROCESS_ DETACH:
            /* do process cleanup */

            /* free TLS index */
            TlsFree( TlsIndex );
            break;
        }
        return( 1 );        /* indicate success */
        /* returning 0 indicates initialization failure */
    }

    EXPORTED void dll_ entry_ 1( void )
    {
        printf( "Hi from dll entry #1\n" );
    }

    EXPORTED void dll_ entry_ 2( void )
    {
        printf( "Hi from dll entry #2\n" );
    }
```

Arguments:

hinstDLL This is a handle for the DLL. It can be used as a argument to other functions such as `GetModuleFileName`.

fdwReason This argument indicates why `LibMain` is being called. It can have one of the following values:

Value ***Meaning***

DLL_PROCESS_ATTACH This value indicates that the DLL is attaching to the address space of the current process as a result of the process starting up or as a result of a call to `LoadLibrary`. A DLL can use this opportunity to initialize any instance data or to use the `TlsAlloc` function to allocate a Thread Local Storage (TLS) index.

During initial process startup or after a call to `LoadLibrary`, the operating system scans the list of loaded DLLs for the process. For each DLL that has not already been called with the `DLL_ PROCESS_ ATTACH` value, the system calls the DLL's `LibMain` entry-point. This call is made in the context of the thread that caused the process address space to change, such as the primary thread of the process or the thread that called `LoadLibrary`.

DLL_THREAD_ATTACH This value indicates that the current process is creating a new thread. When this occurs, the system calls the `LibMain` entry-point of all DLLs currently attached to the process. The call is made in the context of the new thread. DLLs can use this opportunity to initialize a Thread Local Storage (TLS) slot for the thread. A thread calling the DLL's `LibMain` with the `DLL_ PROCESS_ ATTACH` value does not call `LibMain` with the `DLL_ THREAD_ ATTACH` value. Note that `LibMain` is called with this value only by threads created after the DLL is attached to the process.

When a DLL is attached by LoadLibrary, existing threads do not call the LibMain entry-point of the newly loaded DLL.

DLL_THREAD_DETACH This value indicates that a thread is exiting normally. If the DLL has stored a pointer to allocated memory in a TLS slot, it uses this opportunity to free the memory. The operating system calls the LibMain entry-point of all currently loaded DLLs with this value. The call is made in the context of the exiting thread. There are cases in which LibMain is called for a terminating thread even if the DLL never attached to the thread. For example, LibMain is never called with the DLL_ THREAD_ ATTACH value in the context of the thread in either of these two situations:

- The thread was the initial thread in the process, so the system called LibMain with the DLL_ PROCESS_ ATTACH value.
- The thread was already running when a call to the LoadLibrary function was made, so the system never called LibMain for it.

DLL_PROCESS_DETACH This value indicates that the DLL is detaching from the address space of the calling process as a result of either a normal termination or of a call to FreeLibrary. The DLL can use this opportunity to call the TlsFree function to free any TLS indices allocated by using TlsAlloc and to free any thread local data. When a DLL detaches from a process as a result of process termination or as a result of a call to FreeLibrary, the operating system does not call the DLL's LibMain with the DLL_ THREAD_ DETACH value for the individual threads of the process. The DLL is only given DLL_ PROCESS_ DETACH notification. DLLs can take this opportunity to clean up all resources for all threads attached and known to the DLL.

lpvReserved This argument specifies further aspects of DLL initialization and cleanup. If fdwReason is DLL_ PROCESS_ ATTACH,lpvReserved is NULL for dynamic loads and non-NULL for static loads. If fdwReason is DLL_ PROCESS_ DETACH, lpvReserved is NULL if LibMain has been called by using FreeLibrary and non-NULL if LibMain has been called during process termination.

Return Value When the system calls the LibMain function with the DLL_ PROCESS_ ATTACH value, the function returns TRUE (1) if initialization succeeds or FALSE (0) if initialization fails.

If the return value is FALSE (0) when LibMain is called because the process uses the LoadLibrary function, LoadLibrary returns NULL.

If the return value is FALSE (0) when LibMain is called during process initialization, the process terminates with an error. To get extended error information, call GetLastError.

When the system calls LibMain with any value other than DLL_ PROCESS_ ATTACH, the return value is ignored.

Assume the above example is contained in the file dllsamp.c. We can compile the file using the following command. Note that we must specify the "bd" compiler option.

```
C:\>wcc386 -bd dllsamp
```

Before we can link our example, we must create a linker directive file that describes the attributes and entry points of our dynamic link library. The following is a linker directive file, called `dllsamp.lnk`, that can be used to create the dynamic link library.

```
system nt_ dll initinstance terminstance
export dll_ entry_ 1_
export dll_ entry_ 2_
file dllsamp
```

Notes:

1. The "SYSTEM" directive specifies that we are creating a Windows NT dynamic link library.

2. When a dynamic link library uses the Open Watcom C/C++ run-time libraries, an automatic data segment is created each time a new process accesses the dynamic link library. For this reason, initialization code must be executed when a process accesses the dynamic link library for the first time. To achieve this, "INITINSTANCE" must be specified in the "SYSTEM" directive. Similarly, "TERMINSTANCE" must be specified so that the termination code is executed when a process has completed its access to the dynamic link library. If the Open Watcom C/C++ run-time libraries are not used, these options are not required.

3. The "EXPORT" directive specifies the entry points into the dynamic link library. Note that the names specified in the "EXPORT" directive are appended with an underscore. This is the default naming convention used when compiling using the register-based calling convention. No underscore is required when compiling using the stack-based calling convention.

We can now create our dynamic link library by issuing the following command.

```
C:\>wlink @dllsamp
```

A file called `dllsamp.dll` will be created.

32.3 Using Dynamic Link Libraries

It is assumed that all symbols imported by a client application were declared with a `__declspec( dllimport )` modifier when the client application was compiled. At the link stage we have to tell the linker which dynamic libraries the client application should link to. Once we have created a dynamic link library, we must allow other applications to access the functions available in the dynamic link library. There are two ways to achieve this.

The first method is to create a linker directive file which contains an "IMPORT" directive for all entry points in the dynamic link library. The "IMPORT" directive provides the name of the entry point and the name of the dynamic link library. When creating an application that references a function in the dynamic link library, this linker directive file would be included as part of the linking process that created the application.

The second method is to use import libraries. An import library is a standard library that is created from a dynamic link library by using the Open Watcom Library Manager. It contains object modules that describe the entry points in a dynamic link library. The resulting import library can then be specified in a "LIBRARY" directive in the same way one would specify a standard library.

Using an import library is the preferred method of providing references to functions in dynamic link libraries. When a dynamic link library is modified, typically the import library corresponding to the

modified dynamic link library is updated to reflect the changes. Hence, any directive file that specifies the import library in a "LIBRARY" directive need not be modified. However, if you are using "IMPORT" directives, you may have to modify the "IMPORT" directives to reflect the changes in the dynamic link library.

Let us create an import library for our sample dynamic link library we created in the previous section. We do this by issuing the following command.

```
C:\>wlib dllsamp +dllsamp.dll
```

A standard library called `dllsamp.lib` will be created.

Suppose the following sample program, contained in the file `dlltest.c`, calls the functions from our sample dynamic link library.

```
#include <stdio.h>
#include <process.h>
#if defined(__cplusplus)
#define IMPORTED extern "C" __declspec( dllimport )
#else
#define IMPORTED __declspec( dllimport )
#endif

IMPORTED void dll_entry_1( void );
IMPORTED void dll_entry_2( void );
#define STACK_SIZE      8192

static void thread( void *arglist )
{
    printf( "Hi from thread\n" );
    _endthread();
}

int main( void )
{
    unsigned long   tid;

    dll_entry_1();
    tid = _beginthread( thread, STACK_SIZE, NULL );
    dll_entry_2();
    return( 0 );
}
```

We can compile and link our sample application by issuing the following command.

```
C:\>wcl386 -bm -l=nt dlltest dllsamp.lib
```

If we had created a linker directive file of "IMPORT" directives instead of an import library for the dynamic link library, the linker directive file, say `dllimps.lnk`, would be as follows.

```
import dll_entry_1_  dllsamp
import dll_entry_2_  dllsamp
```

Note that the names specified in the "IMPORT" directive are appended with an underscore. This is the default naming convention used when compiling using the register-based calling convention. No underscore is required when compiling using the stack-based calling convention.

To compile and link our sample application, we would issue the following command.

```
C:\>wcl386 -bm -l=nt dlltest -"@dllimps"
```

32.4 The Dynamic Link Library Data Area

The Open Watcom C/C++ 32-bit run-time library does not support the general case operation of DLLs in an execution environment where there is only one instance of the DATA segment (DGROUP) for that DLL.

There are two cases that can lead to a DLL executing with only one instance of the DGROUP.

1. DLLs linked for 32-bit OS/2 without the MANYAUTODATA option.

2. DLLs linked for the Win32 API and executing under Win32s.

In these cases the run-time library startup code detects that there is only one instance of the DGROUP when a second process attempts to attach to the DLL. At that point, it issues a diagnostic for the user and then notifies the operating system that the second process cannot attach to the DLL.

Developers who require DLLs to operate when there is only one instance of the DGROUP can suppress the function which issues the diagnostic and notifies the operating system that the second process cannot attach to the DLL.

Doing so requires good behaviour on the part of processes attaching to the DLL. This good behaviour consists primarily of ensuring that the first process to attach to the DLL is also the last process to detach from the DLL thereby ensuring that the DATA segment is not released back to the free memory pool.

To suppress the function which issues the diagnostic and notifies the operating system that the second process cannot attach to the DLL, the developer must provide a replacement entry point with the following prototype:

```
int __disallow_single_dgroup( int );
```

This function should return zero to indicate that the detected single copy of the DATA segment is allowed.

33 Creating Windows NT POSIX Applications

This chapter describes how to compile and link POSIX applications for Windows NT. There are a number of issues to consider.

1. Open Watcom does not provide its own POSIX libraries. You must use those included with the Microsoft Win32 SDK. They are `libcpsx.lib`, `psxdll.lib` and `psxrtl.lib`. If you installed the Win32 SDK component when you installed the Open Watcom software, you will find these libraries in the `%WATCOM%\lib386\nt` directory.
2. Since you will be using Microsoft POSIX libraries compiled by the Microsoft compiler, you must follow the calling conventions used by Microsoft (i.e., the `__cdecl` convention). The Open Watcom compiler can generate these calling conventions provided that the POSIX library routines are all properly prototyped.
3. Open Watcom does not provide its own header files for use with the Microsoft POSIX libraries. The Microsoft Win32 SDK includes only a subset of the headers required for calling the POSIX library routines. If you installed the Win32 SDK component when you installed the Open Watcom software, you will find these headers in the `%WATCOM%\sdk\posix\h` and `%WATCOM%\sdk\posix\h\sys` directories. Take a look at these directories to see what is and what is not included.
4. If you have the Microsoft compiler, then you will likely have access to the missing header files. If you do not have the Microsoft compiler, then you will have to define prototypes for any of the POSIX library routines that you use for which no prototypes are defined in any of the POSIX header files.
5. There is one exception to the generation of the `__cdecl` calling convention for appropriately prototyped functions. This is the `main` function. Since many Microsoft sample programs inappropriately declare the `main` function as `__cdecl`, it was necessary to make a special case in the Open Watcom compilers to ignore the `__cdecl` attribute when used for this entry point. To work around this problem, a special pragma is used. This is shown in the following example.
6. Since we are going to use the Microsoft POSIX libraries rather than the Open Watcom libraries, we will use the "zl" compile option to instruct the Open Watcom compiler not to include references to Open Watcom libraries in the object files.

To illustrate the creation of a POSIX application, we will use a simple example. This program displays an identifying banner and then displays its arguments one at a time.

Example:

```
[POSIXSMP.C]
#include <unistd.h>

// The Win32 SDK doesn't provide a complete set of
// headers for the libraries (e.g., no stdio.h).

extern int __cdecl printf( char *, ... );

// Note: the "__cdecl" attribute is ignored for main().

int __cdecl main( int argc, char **argv )
{
    int i;
    printf( "POSIX sample program\n" );
    for( i = 0 ; i < argc ; i++ ) {
        printf( "%d: %s\n", i, argv[i] );
    }
    return 0;
}

// Since the "__cdecl" attribute is ignored,
// make sure that parms go on the stack for main
// and that main gets the _  in the right place by
// using a pragma to do so.

#pragma aux main "_*" __parm [];

// The compiler emits references to these symbols,
// so make sure they get defined here to prevent
// unresolved references.

int _cstart_ ;
#pragma aux _cstart_  "*";
int __argc;
#pragma aux __argc "*";
```

The example program illustrates some of the special considerations required for using the Microsoft POSIX libraries rather than the Open Watcom libraries. There are also some special link time issues and these are addressed in the following sample "makefile".

Example:

```
[MAKEFILE]
posixsmp.exe : posixsmp.c posix.add makefile.
  set nt_include=
  set include=$(%watcom)\sdk\posix\h;$(%watcom)\sdk\posix\h\sys
  wcc386 -bt=nt -oaxt -zl posixsmp.c
  wlink @posix.add file posixsmp sys nt_posix option map

posix.add :
  %create posix.add
  %append posix.add system begin nt_posix
  %append posix.add   option osname='Windows NT character-mode posix'
  %append posix.add   libpath %WATCOM%\lib386\nt
  %append posix.add   option nodefaultlib
  %append posix.add   option start=___PosixProcessStartup
  %append posix.add   lib { libcpsx.lib psxrtl.lib psxdll.lib }
  %append posix.add   format windows nt ^
  %append posix.add   runtime posix
  %append posix.add end
```

A new "nt_posix" system is defined in the `posix.add` file. This file is generated automatically by the makefile.

That is about all there is to creating a Windows NT POSIX application. One final note - make sure when using the Microsoft headers that all the library routines that you use are declared as `__cdecl` otherwise your application will not run correctly.

OS/2 Programming Guide

34 Creating 16-bit OS/2 1.x Applications

An OS/2 application can be one of the following; a fullscreen application, a PM-compatible application, or a Presentation Manager application. A fullscreen application runs in its own screen group. A PM-compatible application will run in an OS/2 fullscreen environment or in a window in the Presentation Manager screen group but does not take direct advantage of menus, mouse or other features available in the Presentation Manager. A Presentation Manager application has full access to the complete set of user-interface tools such as menus, icons, scroll bars, etc.

This chapter deals with the creation of OS/2 fullscreen applications. For information on creating Presentation Manager applications, refer to the section entitled "Programming for OS/2 Presentation Manager" on page 257.

We will illustrate the steps to creating 16-bit OS/2 1.x applications by taking a small sample application and showing you how to compile, link, run and debug it.

34.1 The Sample Application

To demonstrate the creation of 16-bit OS/2 1.x applications using command-line oriented tools, we introduce a simple sample program. For our example, we are going to use the famous "hello" program.

```
#include <stdio.h>

void main()
{
    printf( "Hello world\n" );
}
```

The C++ version of this program follows:

```
#include <iostream.h>

void main()
{
    cout << "Hello world" << endl;
}
```

The goal of this program is to display the message "Hello world" on the screen. The C version uses the C library `printf` routine to accomplish this task. The C++ version uses the "iostream" library to accomplish this task. We will take you through the steps necessary to produce this result.

34.2 Building and Running the Sample OS/2 1.x Application

To compile and link our example program which is stored in the file `hello.c`, enter the following command:

```
[C:\]wcl -l=os2 hello.c
```

The typical messages that appear on the screen are shown in the following illustration.

```
[C:\]wcl -l=os2  hello.c
Open Watcom C/C++16 Compile and Link Utility
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1988-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
        wcc hello.c
Open Watcom C16 Optimizing Compiler
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1984-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
hello.c: 6 lines, included 155, 0 warnings, 0 errors
Code size: 17

Open Watcom Linker
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1985-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
loading object files
searching libraries
creating an OS/2 16-bit executable
```

Provided that no errors were encountered during the compile or link phases, the "hello" program may now be run.

```
[C:\]hello
Hello world
```

If you examine the current directory, you will find that two files have been created. These are `hello.obj` (the result of compiling `hello.c`) and `hello.exe` (the result of linking `hello.obj` with the appropriate Open Watcom C/C++ libraries). It is `hello.exe` that is run by OS/2 when you enter the "hello" command.

34.3 Debugging the Sample OS/2 1.x Application

Let us assume that you wish to debug your application in order to locate an error in programming. In the previous section, the "hello" program was compiled with default compile and link options. When debugging an application, it is useful to refer to the symbolic names of routines and variables. It is also convenient to debug at the source line level rather than the machine language level. To do this, we must direct both the compiler and linker to include additional debugging information in the object and executable files. Using the `WCL` command, this is fairly straightforward. `WCL` recognizes the Open Watcom C/C++ compiler "debug" options and will create the appropriate debug directives for the Open Watcom Linker.

For example, to compile and link the "hello" program with debugging information, the following command may be issued.

```
[C:\]wcl -l=os2 -d2 hello.c
```

The typical messages that appear on the screen are shown in the following illustration.

```
[C:\]wcl -l=os2  -d2 hello.c
Open Watcom C/C++16 Compile and Link Utility
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1988-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
        wcc hello.c  -d2
Open Watcom C16 Optimizing Compiler
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1984-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
hello.c: 6 lines, included 155, 0 warnings, 0 errors
Code size: 23

Open Watcom Linker
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1985-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
loading object files
searching libraries
creating an OS/2 16-bit executable
```

The "d2" option requests the maximum amount of debugging information that can be provided by the Open Watcom C/C++ compiler. `WCL` will make sure that this debugging information is included in the executable file that is produced by the linker.

The "Code size" value is larger than in the previous example since selection of the "d2" option results in fewer code optimizations by default. You can request more optimization by specifying the appropriate options. However, you do so at the risk of making it more difficult for yourself to determine the relationship between the object code and the original source language code.

For OS/2, you should also include the `BINP\DLL` directory in the "LIBPATH" directive of the system configuration file `CONFIG.SYS`. It contains the Open Watcom Debugger Dynamic Link Libraries (DLLs).

Example:

```
libpath=c:\watcom\binp\dll
```

To request the Open Watcom Debugger to assist in debugging the application, the following command may be issued.

```
[C:\]wd hello
```

It would be too ambitious to describe the debugger in this introductory chapter so we refer you to the book entitled ***Open Watcom Debugger User's Guide.***

35 Creating 32-bit OS/2 Applications

An OS/2 application can be one of the following; a fullscreen application, a PM-compatible application, or a Presentation Manager application. A fullscreen application runs in its own screen group. A PM-compatible application will run in an OS/2 fullscreen environment or in a window in the Presentation Manager screen group but does not take direct advantage of menus, mouse or other features available in the Presentation Manager. A Presentation Manager application has full access to the complete set of user-interface tools such as menus, icons, scroll bars, etc.

This chapter deals with the creation of OS/2 fullscreen applications. For information on creating Presentation Manager applications, refer to the section entitled "Programming for OS/2 Presentation Manager" on page 257.

We will illustrate the steps to creating 32-bit OS/2 applications by taking a small sample application and showing you how to compile, link, run and debug it.

35.1 The Sample Application

To demonstrate the creation of 32-bit OS/2 applications using command-line oriented tools, we introduce a simple sample program. For our example, we are going to use the famous "hello" program.

```
#include <stdio.h>

void main()
{
    printf( "Hello world\n" );
}
```

The C++ version of this program follows:

```
#include <iostream.h>

void main()
{
    cout << "Hello world" << endl;
}
```

The goal of this program is to display the message "Hello world" on the screen. The C version uses the C library `printf` routine to accomplish this task. The C++ version uses the "iostream" library to accomplish this task. We will take you through the steps necessary to produce this result.

35.2 Building and Running the Sample OS/2 Application

To compile and link our example program which is stored in the file `hello.c`, enter the following command:

```
[C:\]wcl386 -l=os2v2 hello.c
```

The typical messages that appear on the screen are shown in the following illustration.

```
[C:\]wcl386 -l=os2v2  hello.c
Open Watcom C/C++32 Compile and Link Utility
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1988-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
        wcc386 hello.c
Open Watcom C32 Optimizing Compiler
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1984-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
hello.c: 6 lines, included 174, 0 warnings, 0 errors
Code size: 24

Open Watcom Linker
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1985-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
loading object files
searching libraries
creating an OS/2 32-bit executable
```

Provided that no errors were encountered during the compile or link phases, the "hello" program may now be run.

```
[C:\]hello
Hello world
```

If you examine the current directory, you will find that two files have been created. These are `hello.obj` (the result of compiling `hello.c`) and `hello.exe` (the result of linking `hello.obj` with the appropriate Open Watcom C/C++ libraries). It is `hello.exe` that is run by OS/2 when you enter the "hello" command.

35.3 Debugging the Sample OS/2 Application

Let us assume that you wish to debug your application in order to locate an error in programming. In the previous section, the "hello" program was compiled with default compile and link options. When debugging an application, it is useful to refer to the symbolic names of routines and variables. It is also convenient to debug at the source line level rather than the machine language level. To do this, we must direct both the compiler and linker to include additional debugging information in the object and executable files. Using the `WCL386` command, this is fairly straightforward. `WCL386` recognizes the Open Watcom C/C++ compiler "debug" options and will create the appropriate debug directives for the Open Watcom Linker.

For example, to compile and link the "hello" program with debugging information, the following command may be issued.

```
[C:\]wcl386 -l=os2v2 -d2 hello.c
```

The typical messages that appear on the screen are shown in the following illustration.

```
[C:\]wcl386 -l=os2v2  -d2 hello.c
Open Watcom C/C++32 Compile and Link Utility
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1988-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
        wcc386 hello.c  -d2
Open Watcom C32 Optimizing Compiler
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1984-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
hello.c: 6 lines, included 174, 0 warnings, 0 errors
Code size: 45

Open Watcom Linker
Copyright (c) 2002-2019 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1985-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
loading object files
searching libraries
creating an OS/2 32-bit executable
```

The "d2" option requests the maximum amount of debugging information that can be provided by the Open Watcom C/C++ compiler. `WCL386` will make sure that this debugging information is included in the executable file that is produced by the linker.

The "Code size" value is larger than in the previous example since selection of the "d2" option results in fewer code optimizations by default. You can request more optimization by specifying the appropriate options. However, you do so at the risk of making it more difficult for yourself to determine the relationship between the object code and the original source language code.

To request the Open Watcom Debugger to assist in debugging the application, the following command may be issued.

```
[C:\]wd hello
```

It would be too ambitious to describe the debugger in this introductory chapter so we refer you to the book entitled ***Open Watcom Debugger User's Guide.***

36 OS/2 2.x Multi-threaded Applications

This chapter describes how to create multi-threaded applications. A multi-threaded application is one whose tasks are divided among several threads of execution. A process is an executing application and the resources it uses. A thread is the smallest unit of execution within a process. Each thread has its own stack and a set of machine registers and shares all resources with its parent process. The path of execution of one thread does not affect that of another; each thread is an independent entity.

Typically, an application has a single thread of execution. In this type of application, all tasks, once initiated, are completed before the next task begins. In contrast, tasks in a multi-threaded application can be performed concurrently since more than one thread is executing at once. For example, each thread may be designed to perform a separate task.

36.1 Programming Considerations

Since a multi-threaded application consists of many threads of execution, there are a number of issues that you must consider.

Since threads share the resources of its parent, it may be necessary to serialize access to these resources. For example, if your application has a function that displays information on the console and is used by all threads, it is necessary to allow only one thread to use that function at any time. That is, once a thread calls that function, the function should ensure that no other thread displays information until all information for the initial thread has been displayed. An example of such a function is the `printf` library function.

Another issue that must be considered when creating multi-threaded applications is global variables. If you have global variables that contain thread-specific information, there must be an instance of each global variable for each thread. An example of such a variable is the `errno` global variable defined in the run-time libraries. If an error condition was created by a thread, you would not want it to affect the execution of other threads. Therefore, each thread should contain its own instance of this variable.

36.2 Creating Threads

Each application initially contains a single thread. The run-time libraries contain two functions that create and terminate threads of execution. The function `_ beginthread` creates a thread of execution and the function `_ endthread` ends a thread of execution. The macro`_ threadid` can be used to determine the current thread identifier.

> ***WARNING!*** If any thread calls a library function, you must use the `_beginthread` function to create the thread. Do not use the `DosCreateThread` API function.

36.2.1 Creating a New Thread

The `_beginthread` function creates a new thread. It is defined as follows.

```
int _beginthread( void (*start_address)(void *),
                  void *stack_bottom,
                  unsigned stack_size,
                  void *arglist );
```

where ***description:***

start_address is the address of the function that will be called when the newly created thread is executed. When the thread returns from that function, the thread will be terminated. Note that a call to the `_endthread` function will also terminate the thread.

stack_bottom specifies the bottom of the stack to be used by the thread. Note that this argument is ignored as it is only needed to simplify the port of OS/2 1.x multi-threaded applications to OS/2 2.x. Under OS/2 2.x, the operating system allocates the stack for the new thread. A value of NULL may be specified.

stack_size specifies the size of the stack to be allocated by the operating system for the new thread. The stack size should be a multiple of 4K.

arglist is passed as an argument to the function specified by `start_address`. If no argument is required, a value of NULL can be specified.

If a new thread is successfully created, the thread identifier of the new thread is returned. Otherwise, a value of -1 is returned.

The header file `process.h` contains the definition of the `_beginthread` function.

36.2.2 Terminating the Current Thread

The `_endthread` function terminates the current thread. It is defined as follows.

```
void _endthread( void )
```

The header file `process.h` contains the definition of the `_endthread` function.

36.2.3 Getting the Current Thread Identifier

The `_threadid` macro can be used to determine the current thread identifier. It is defined as follows.

```
int *__threadid(void);
#define _threadid (__threadid())
```

The header file `stddef.h` contains the definition of the `_threadid` macro.

36.3 A Multi-threaded Example

Let us create a simple multi-threaded application. The source code for this example can be found in `\watcom\samples\os2.`

```
#include <process.h>
#include <stdio.h>
#include <stddef.h>
#define INCL_DOS
#include <os2.h>

static  volatile int    NumThreads;
static  volatile int    HoldThreads;

#define NUM_THREADS     5
#define STACK_SIZE      32768

static void a_thread( void *arglist )
/***********************************/
{
    while( HoldThreads ) {
        DosSleep( 1 );
    }
    printf( "Hi from thread %d\n", *_threadid );
    DosEnterCritSec();
    --NumThreads;
    DosExitCritSec();
    _endthread();
}

int main( void )
/**************/
{
    int         i;

    printf( "Initial thread id = %d\n", *_threadid );
    NumThreads = 0;
    HoldThreads = 1;
    /* initial thread counts as 1 */
    for( i = 2; i <= NUM_THREADS; ++i ) {
        if( _beginthread( a_thread, NULL, STACK_SIZE, NULL ) == -1 ) {
            printf( "creation of thread %d failed\n", i );
        } else {
            ++NumThreads;
        }
    }
    HoldThreads = 0;
    while( NumThreads != 0 ) {
        DosSleep( 1 );
    }
    return( 0 );
}
```

Note:

1. In the function `a_thread`,`DosEnterCritSec` and `DosExitCritSec` are called when we modify the variable `NumThreads`. This ensures that the action of extracting the value of `NumThreads` from memory, incrementing the value, and storing the new result into memory, occurs without interruption. If these functions were not called, it would be possible for two threads to extract the value of `NumThreads` from memory before an update occurred.

Let us assume that the file `mthread.c` contains the above example. Before compiling the file, make sure that the **WATCOM** environment variable is set to the directory in which you installed Open Watcom C/C++. Also, the **INCLUDE** environment variable must include the `\watcom\h\os2` and `\watcom\h` directories ("\WATCOM" is the directory in which Open Watcom C/C++ was installed).

We can now compile and link the application by issuing the following command.

```
[C:\]wcl386 -bt=os2 -bm -l=os2v2 mthread
```

The "bm" option must be specified since we are creating a multi-threaded application. If your multi-threaded application contains more than one module, each module must be compiled using the "bm" switch.

The "l" option specifies the target system for which the application is to be linked. The system name `os2v2` is defined in the file `wlsystem.lnk` which is located in the "BINW" subdirectory of the directory in which you installed Open Watcom C/C++.

The multi-threaded application is now ready to be run.

36.4 Thread Limits

There is a limit to the number of threads an application can create under 16-bit OS/2. The default limit is 32. This limit can be adjusted by statically initializing the unsigned global variable `__MaxThreads`.

Under 32-bit OS/2, there is no limit to the number of threads an application can create. However, due to the way in which multiple threads are supported in the Open Watcom libraries, there is a small performance penalty once the number of threads exceeds the default limit of 32 (this number includes the initial thread). If you are creating more than 32 threads and wish to avoid this performance penalty, you can redefine the threshold value of 32. You can statically initialize the global variable `__MaxThreads`.

By adding the following line to your multi-threaded application, the new threshold value will be set to 48.

```
unsigned __MaxThreads = { 48 };
```

37 OS/2 2.x Dynamic Link Libraries

A dynamic link library, like a standard library, is a library of functions. When an application uses functions from a standard library, the library functions referenced by the application become part of the executable module. This form of linking is called static linking. When an application uses functions from a dynamic link library, the library functions referenced by the application are not included in the executable module. Instead, the executable module contains references to these functions which are resolved when the application is loaded. This form of linking is called dynamic linking.

Let us consider some of the advantages of using dynamic link libraries over standard libraries.

1. Functions in dynamic link libraries are not linked into your program. Only references to the functions in dynamic link libraries are placed in the program module. These references are called import definitions. As a result, the linking time is reduced and disk space is saved. If many applications reference the same dynamic link library, the saving in disk space can be significant.

2. Since program modules only reference dynamic link libraries and do not contain the actual executable code, a dynamic link library can be updated without re-linking your application. When your application is executed, it will use the updated version of the dynamic link library.

3. Dynamic link libraries also allow sharing of code and data between the applications that use them. If many applications that use the same dynamic link library are executing concurrently, the sharing of code and data segments improves memory utilization.

37.1 Creating Dynamic Link Libraries

Once you have developed the source for a library of functions, a number of steps are required to create a dynamic link library containing those functions.

First, you must compile your source using the "bd" compiler option. This option tells the compiler that the module you are compiling is part of a dynamic link library. Once you have successfully compiled your source, you must create a linker directive file that describes the attributes of your dynamic link library. The following lists the most common linker directives required to create a dynamic link library.

1. The "SYSTEM" directive is used to specify that a dynamic link library is to be created.
2. The "EXPORT" directive is used to to specify which functions in the dynamic link library are to be exported.

Specifying exports in the source code

The "EXPORT" directive need not be used when the symbols to be exported are declared with the `__export` type qualifier in the source code. Such symbols are exported automatically, through special records inserted into the object files by the compiler.

3. The "OPTION" directive is used to specify attributes such as the name of the dynamic link library and how to allocate the automatic data segment when the dynamic link library is referenced.
4. The "SEGMENT" directive is used to specify attributes of segments. For example, a segment may be read-only or read-write.

Once the dynamic link library is created, you must allow access to the dynamic link library to client applications that wish to use it. This can be done by creating an import library for the dynamic link library or creating a linker directive file that contains "IMPORT" directives for each of the entry points in the dynamic link library.

37.2 Creating a Sample Dynamic Link Library

Let us now create a dynamic link library using the following example.

```
#include <stdio.h>
#include <os2.h>

#if defined(__cplusplus)
#define EXTERNC extern "C"
#else
#define EXTERNC
#endif

unsigned APIENTRY LibMain( unsigned hmod, unsigned termination )
{
    if( termination ) {
        /* DLL is detaching from process */
    } else {
        /* DLL is attaching to process */
    }
    return( 1 );
}

EXTERNC void dll_entry_1( void )
{
    printf( "Hi from dll entry #1\n" );
}

EXTERNC void dll_entry_2( void )
{
    printf( "Hi from dll entry #2\n" );
}
```

32-bit OS/2 DLLs can include a `LibMain` entry point when you are using the Open Watcom C/C++ run-time libraries.

Arguments:

hmod This is a handle for the DLL.

termination A 0 value indicates that the DLL is attaching to the address space of the current process as a result of the process starting up or as a result of a call to `DosLoadModule`. A DLL can use this opportunity to initialize any instance data.

A non-zero value indicates that the DLL is detaching from the address space of the calling process as a result of either a normal termination or of a call to `DosFreeModule`.

Return Value The `LibMain` function returns 1 if initialization succeeds or 0 if initialization fails.

If the return value is 0 when `LibMain` is called because the process uses the `DosLoadModule` function, `DosLoadModule` returns an error.

If the return value is 0 when `LibMain` is called during process initialization, the process terminates with an error.

Assume the above example is contained in the file `dllsamp.c`. We can compile the file using the following command. Note that we must specify the "bd" compiler option.

```
[C:\]wcc386 -bd dllsamp
```

Before we can link our example, we must create a linker directive file that describes the attributes and entry points of our dynamic link library. The following is a linker directive file, called `dllsamp.lnk`, that can be used to create the dynamic link library.

```
system os2v2 dll initinstance terminstance
option manyautodata
export dll_ entry_ 1_
export dll_ entry_ 2_
file dllsamp
```

Notes:

1. The "SYSTEM" directive specifies that we are creating a 32-bit OS/2 dynamic link library.

2. The "MANYAUTODATA" option specifies that the automatic data segment is allocated for every instance of the dynamic link library. This option must be specified only for a dynamic link library that uses the Open Watcom C/C++ run-time libraries. If the Open Watcom C/C++ run-time libraries are not used, this option is not required. Our example does use the Open Watcom C/C++ run-time libraries so we must specify the "MANYAUTODATA" option.

   As was just mentioned, when a dynamic link library uses the Open Watcom C/C++ run-time libraries, an automatic data segment is created each time a process accesses the dynamic link library. For this reason, initialization code must be executed when a process accesses the dynamic link library for the first time. To achieve this, "INITINSTANCE" must be specified in the "SYSTEM" directive. Similarly, "TERMINSTANCE" must be specified so that the termination code is executed when a process has completed its access to the dynamic link library. If the Open Watcom C/C++ run-time libraries are not used, these options are not required.

3. The "EXPORT" directive specifies the entry points into the dynamic link library. Note that the names specified in the "EXPORT" directive are appended with an underscore. This is the default naming convention used when compiling using the register-based calling convention. No underscore is required when compiling using the stack-based calling convention.

We can now create our dynamic link library by issuing the following command.

```
[C:\]wlink @dllsamp
```

A file called `dllsamp.dll` will be created.

37.3 Using Dynamic Link Libraries

Once we have created a dynamic link library, we must allow other applications to access the functions available in the dynamic link library. There are two ways to achieve this.

The first method is to create a linker directive file which contains an "IMPORT" directive for all entry points in the dynamic link library. The "IMPORT" directive provides the name of the entry point and the name of the dynamic link library. When creating an application that references a function in the dynamic link library, this linker directive file would be included as part of the linking process that created the application.

The second method is to use import libraries. An import library is a standard library that is created from a dynamic link library by using the Open Watcom Library Manager. It contains object modules that describe the entry points in a dynamic link library. The resulting import library can then be specified in a "LIBRARY" directive in the same way one would specify a standard library.

Using an import library is the preferred method of providing references to functions in dynamic link libraries. When a dynamic link library is modified, typically the import library corresponding to the modified dynamic link library is updated to reflect the changes. Hence, any directive file that specifies the import library in a "LIBRARY" directive need not be modified. However, if you are using "IMPORT" directives, you may have to modify the "IMPORT" directives to reflect the changes in the dynamic link library.

Let us create an import library for our sample dynamic link library we created in the previous section. We do this by issuing the following command.

```
[C:\]wlib dllsamp +dllsamp.dll
```

A standard library called `dllsamp.lib` will be created.

Suppose the following sample program, contained in the file `dlltest.c`, calls the functions from our sample dynamic link library.

```
#include <stdio.h>
#if defined(__cplusplus)
#define EXTERNC extern "C"
#else
#define EXTERNC
#endif

EXTERNC void dll_entry_1( void );
EXTERNC void dll_entry_2( void );
int main( void )
{
    dll_entry_1();
    dll_entry_2();
    return( 0 );
}
```

We can compile and link our sample application by issuing the following command.

```
[C:\]wcl386 -l=os2v2 dlltest dllsamp.lib
```

If we had created a linker directive file of "IMPORT" directives instead of an import library for the dynamic link library, the linker directive file, say `dllimps.lnk`, would be as follows.

```
import dll_entry_1_  dllsamp
import dll_entry_2_  dllsamp
```

Note that the names specified in the "IMPORT" directive are appended with an underscore. This is the default naming convention used when compiling using the register-based calling convention. No underscore is required when compiling using the stack-based calling convention.

To compile and link our sample application, we would issue the following command.

```
[C:\]wcl386 -l=os2v2 dlltest -"@dllimps"
```

37.4 The Dynamic Link Library Data Area

The Open Watcom C/C++ 32-bit run-time library does not support the general case operation of DLLs in an execution environment where there is only one instance of the DATA segment (DGROUP) for that DLL.

There are two cases that can lead to a DLL executing with only one instance of the DGROUP.

1. DLLs linked for 32-bit OS/2 without the MANYAUTODATA option.
2. DLLs linked for the Win32 API and executing under Win32s.

In these cases the run-time library startup code detects that there is only one instance of the DGROUP when a second process attempts to attach to the DLL. At that point, it issues a diagnostic for the user and then notifies the operating system that the second process cannot attach to the DLL.

Developers who require DLLs to operate when there is only one instance of the DGROUP can suppress the function which issues the diagnostic and notifies the operating system that the second process cannot attach to the DLL.

Doing so requires good behaviour on the part of processes attaching to the DLL. This good behaviour consists primarily of ensuring that the first process to attach to the DLL is also the last process to detach from the DLL thereby ensuring that the DATA segment is not released back to the free memory pool.

To suppress the function which issues the diagnostic and notifies the operating system that the second process cannot attach to the DLL, the developer must provide a replacement entry point with the following prototype:

```
int __disallow_single_dgroup( int );
```

This function should return zero to indicate that the detected single copy of the DATA segment is allowed.

37.5 Dynamic Link Library Initialization/Termination

Each dynamic link library (DLL) has an initialization and termination routine associated with it. The initialization routine can either be called the first time any process accesses the DLL ("INITGLOBAL" is specified at link time) or each time a process accesses the DLL ("INITINSTANCE" is specified at link time). Similarly, the termination routine can either be called when all processes have completed their access of the DLL ("TERMGLOBAL" is specified at link time) or each time a process completes its access of the DLL ("TERMINSTANCE" is specified at link time).

For a DLL that uses the C/C++ run-time libraries, initialization and termination of the C/C++ run-time environment is performed automatically. It is also possible for a DLL to do its own special initialization and termination process.

The C/C++ run-time environment provides two methods for calling user-written DLL initialization and termination code.

1. If you provide your own version of `LibMain` then it will be called for initialization and termination. The use of `LibMain` is described earlier in this chapter.

2. If you do not provide your own version of `LibMain` then a default version is linked in from the library. This version will call `__dll_initialize` for DLL initialization and `__dll_terminate` for DLL termination. Default stub versions of these two routines are included in the run-time library. If you wish to perform additional initialization/termination that is specific to your dynamic link library, you may write your own versions of these routines.

Once the C/C++ run-time environment is initialized, the routine `__dll_initialize` is called. After the C/C++ run-time environment is terminated, the routine `__dll_terminate` is called. This last point is important since it means that you cannot do any run-time calls in the termination routine.

The initialization and termination routines return an integer. A value of 0 indicates failure; a value of 1 indicates success. The following example illustrates sample initialization/termination routines.

```
#include <stdlib.h>

#define WORKING_SIZE (64 * 1024)

char *WorkingStorage;

#if defined(__cplusplus)
#define EXTERNC extern "C"
#else
#define EXTERNC
#endif

void __dll_finalize( void );

EXTERNC int __dll_initialize( void )
{
    WorkingStorage = malloc( WORKING_SIZE );
    if( WorkingStorage == NULL ) return( 0 );
    atexit( __dll_finalize );
    return( 1 );
}

void __dll_finalize( void )
{
    free( WorkingStorage );
}

EXTERNC int __dll_terminate( void )
{
    return( 1 );
}

EXTERNC void dll_entry( void )
{
    /* use array WorkingStorage */
}
```

In the above example, the process initialization routine allocates storage that the dynamic link library needs, the routine `dll_entry` uses the storage, and the process termination routine frees the storage allocated in the initialization routine.

38 Programming for OS/2 Presentation Manager

Basically, there are two classes of C/C++ applications that can run in a windowed environment.

The first are those C/C++ applications that do not use any of the Presentation Manager API functions; they are strictly C/C++ applications that do not rely on the features of a particular operating system.

The second class of C/C++ applications are those that actually call Presentation Manager API functions directly. These are applications that have been tailored for the Presentation Manager operating environment.

It is assumed that the reader is familiar with the concepts of Presentation Manager programming.

38.1 Porting Existing C/C++ Applications

Suppose you have a set of C/C++ applications that previously ran under DOS and you now wish to run them under OS/2. To achieve this, simply recompile your application and link with the appropriate libraries. Depending on the method with which you linked your application, it can run in an OS/2 fullscreen environment, a PM-compatible window, or as a Presentation Manager application. An OS/2 fullscreen application runs in its own screen group. A PM-compatible application will run in an OS/2 fullscreen environment or in a window in the Presentation Manager screen group but does not take direct advantage of menus, mouse or other features available in the Presentation Manager. A Presentation Manager application has full access to the complete set of user-interface tools such as menus, icons, scroll bars, etc. However, porting a console oriented application to Presentation Manager often requires significant effort and a substantial redesign of the application.

38.1.1 An Example

Very little effort is required to port an existing C/C++ application to OS/2. Let us try to run the following sample program (contained in the file `hello.c`).

```
#include <stdio.h>

int main( void )
/***************/
{
    printf( "Hello world!\n" );
    return( 0 );
}
```

An equivalent C++ program follows:

```
#include <iostream.h>

int main( void )
{
    cout << "Hello world" << endl;
    return( 0 );
}
```

First we must compile the file `hello.c` by issuing the following command.

```
[C:\]wcc386 hello
```

Once we have successfully compiled the file, we can link it by issuing the following command.

```
[C:\]wlink sys os2v2 file hello
```

It is also possible to compile and link in one step, by issuing the following command.

```
[C:\]wcl386 -l=os2v2 hello
```

This will create a PM-compatible application. If you wish to create a fullscreen application, link with the following command.

```
[C:\]wlink sys os2v2 fullscreen file hello
```

38.2 Calling Presentation Manager API Functions

It is also possible for a C/C++ application to create its own windowing environment. This is achieved by calling PM API functions directly from your C/C++ program. The techniques for developing these applications can be found in the ***OS/2 Technical Library.***

A number of C/C++ include files (files with extension `.h`) are provided which define Presentation Manager data structures and constants. They are located in the `\watcom\h\os2` directory. These include files are roughly equivalent to the C/C++ header files that are available with the IBM OS/2 Developer's Toolkit.

A sample C/C++ Presentation Manager application is also located in the `\watcom\samples\os2` directory. It is contained in the files `shapes.c` (C variant) and `shapes.cpp` (C++ variant, nearly identical). The file `shapes.c` contains the following.

```
#include <stdlib.h>

#define INCL_WIN
#define INCL_GPI
#include <os2.h>

int             SizeX;
int             SizeY;
HWND            FrameHandle;
HMQ             hMessageQueue;
HAB             AnchorBlock;
```

```
static int      Random( int high )
{
    return( ( (double)rand() / 32767 ) * high );
}

static void NewColor( HPS ps )
{
    GpiSetColor( ps, Random( 15 ) + 1 );
}

/* Draw a rectangular shape of random size and color at random position */
static void DrawEllipse(HWND hwndWindow)
{
    POINTL      ptl;
    HPS         ps;
    static int  Odd = 0;
    int         parm1,parm2;

    ps = WinGetPS( hwndWindow );
    ptl.x = Random( SizeX );
    ptl.y = Random( SizeY );
    GpiMove( ps, &ptl );
    ptl.x = Random( SizeX );
    ptl.y = Random( SizeY );
    parm1 = Random( 50 );
    parm2 = Random( 50 );
    if( Random( 10 ) >= 5 ) {
        NewColor( ps );
        GpiBox( ps, DRO_ FILL, &ptl, 0, 0 );
        NewColor( ps );
        GpiBox( ps, DRO_ OUTLINE, &ptl, 0, 0 );
    } else {
        NewColor( ps );
        GpiBox( ps, DRO_ FILL, &ptl, parm1, parm2 );
        NewColor( ps );
        GpiBox( ps, DRO_ OUTLINE, &ptl, parm1, parm2 );
    }
    Odd++;
    Odd &= 1;
    WinReleasePS( ps );
}

/* Client window procedure */
MRESULT EXPENTRY MainDriver( HWND hwnd, USHORT msg, MPARAM mp1, MPARAM mp2 )
{
    HPS         ps;
    RECTL       rcl;

    switch( msg ) {
    case WM_ CREATE:
        /* Start a 150ms timer on window creation */
        WinStartTimer( AnchorBlock, hwnd, 1, 150 ) ;
        break;
    case WM_ TIMER:
        /* Draw another ellipse on each timer tick */
        DrawEllipse( hwnd );
        return( 0 );
    case WM_ SIZE:
        /* Remember new dimensions when window is resized */
        SizeX = SHORT1FROMMP( mp2 );
        SizeY = SHORT2FROMMP( mp2 );
        return( 0 );
    case WM_ PAINT:
        /* Handle paint events */
        ps = WinBeginPaint( hwnd, NULL, NULL );
        WinQueryWindowRect( hwnd, &rcl );
        WinFillRect( ps, &rcl, CLR_ WHITE );
        WinEndPaint( ps );
        return( 0 );
```

```
    }
    /* Let the default window procedure handle all other messages */
    return( WinDefWindowProc( hwnd, msg, mp1, mp2 ) );
}

int     main()
{
    ULONG       style;
    QMSG        qmsg;
    HWND        WinHandle;

    /* Initialize windowing and create message queue */
    AnchorBlock = WinInitialize( 0 );
    if( AnchorBlock == 0 ) return( 0 );
    hMessageQueue = WinCreateMsgQueue( AnchorBlock, 0 );
    if( hMessageQueue == 0 ) return( 0 );

    /* Register window class */
    if( !WinRegisterClass( AnchorBlock, "Watcom", (PFNWP)MainDriver,
                           CS_SIZEREDRAW, 0 ) ) {
        return( 0 );
    }

    /* Create frame and client windows */
    style = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER | FCF_MINMAX |
            FCF_SHELLPOSITION | FCF_TASKLIST;
    FrameHandle = WinCreateStdWindow( HWND_DESKTOP, WS_VISIBLE, &style,
                                      "Watcom",
                                      "Shapes - C sample",
                                      0, NULL, 0, &WinHandle );

    /* If window creation failed, exit immediately! */
    if( FrameHandle == 0 ) return( 0 );

    /* Message loop */
    while( WinGetMsg( AnchorBlock, &qmsg, NULL, 0, 0 ) ) {
        WinDispatchMsg( AnchorBlock, &qmsg );
    }

    /* Shut down and clean up */
    WinDestroyWindow( FrameHandle );
    WinDestroyMsgQueue( hMessageQueue );
    WinTerminate( AnchorBlock );

    return( 1 );
}
```

You can compile, link and run this demonstration by issuing the following commands.

```
[C:\]wcl386 -l=os2v2_pm shapes
[C:\]shapes
```

39 Developing an OS/2 Physical Device Driver

In this chapter, we discuss the development of Physical Device Drivers (PDD) for OS/2. The tools used in the creation of the sample PDD are:

- the 16-bit Open Watcom C compiler
- the Open Watcom Assembler
- the Open Watcom Make utility

The sample Physical Device Driver that we are going to build, `HRTIMER.SYS`, provides access to a high resolution timer. Additional sources of information on PDDs can be found in the following:

1. ***OS/2 2.0 Technical Library - Physical Device Driver Reference***
2. ***Writing OS/2 2.1 Device Drivers in C*** by Steve J. Mastrianni
3. ***An OS/2 High Resolution Software Timer*** by Derek Williams, an article which appeared in the Fall 1991 issue of ***IBM Personal Systems Developer*** magazine. The source code for this device driver was adapted from the magazine article. For detailed information on the way this device driver works, please read that article.

`HRTIMER.SYS` is a 16-bit device driver which runs under OS/2 1.x and 2.x/3.x. It has a resolution of 840 nanoseconds (i.e., 1 tick of the Intel 8253/8254 timer = 840 nanoseconds).

Here are some notes on creating Physical Device Drivers using Open Watcom software tools.

1. A Physical Device Driver is linked as a DLL.
2. The first segment must be a data segment, the next a code segment.
3. By default only the first two segments remain after initialization, extra segments have to be marked IOPL.
4. The assembler file, `DEVSEGS.ASM`, defines the segment ordering.
5. `#pragma dataseg` and `#pragma codeseg` are used to get various pieces of code and data into the correct segments.
6. The `_ HEADER` segment contains the device header and must be at the beginning of the data segment.
7. The `_ INITCODE` and`_ INITDATA` segments are used to place initialization code and data at the end so it can be discarded by OS/2.

To compile the source code for the 16-bit Physical Device Drivers, we use the following options:

-bt=os2 build target is OS/2

-ms 16-bit small code/small data model

-5 Pentium optimizations (this is optional)

-omi inline math and intrinsic functions (this is optional)

-s no stack checking

-zdp DS is pegged to DGROUP

-zff FS floats, i.e. not fixed to a segment

-zgf GS floats, i.e. not fixed to a segment

-zu SS != DGROUP

-zl remove default library information

To link the object code for the 16-bit Physical Device Drivers, we use the following options:

name hrtimer.sys to name the executable file.

sys os2 dll initglobal to link a 16-bit OS/2 DLL. Specifying INITGLOBAL will cause the initialization routine to be called the first time the dynamic link library is loaded.

option map to generate a map file.

option quiet to minimize the number of linker informational messages.

lib os2 to include the 16-bit `OS2.LIB` library file.

file ... to include the component object files of the device driver.

The sample files used to create the Physical Device Driver and the programs that use it are located in the `\WATCOM\SRC\OS2\PDD` directory. The Physical Device Driver files are:

DEVSEGS.ASM This small assembler file orders the segment definitions in the executable file.

```
Data Segments   _HEADER
                CONST
                CONST2
                _DATA
                _BSS
                _INITDATA (discardable)
Code Segments   _TEXT
                _INITCODE (discardable)
```

HEADER.C The first thing that must follow the EXE Header is the Device Driver Header.

STRATEGY.C This is the resident portion of the Strategy routine.

STRATINI.C This is the discardable portion of the Strategy routine, the initialization code and data.

HRTIMER.H This file contains the definition of the timer "timestamp" structure.

HRDEV.H This file contains definitions for the Intel 8253 hardware timer.

DEVHDR.H This file contains definitions for the Device Driver Header structure (see page 3-2, "Physical Device Driver Header" of PDD Reference).

DEVDEFS.H This file provides type definitions.

DEVREQP.H This file contains definitions for the Device Driver Request Packets.

DEVAUX.H This file contains definitions for the Device Driver Help (DevHlp) routines.

The demonstration program files are:

HRTEST.C This file is a sample C program that shows how to use the device driver to calculate elapsed times. It demonstrates how to open the device driver, read timestamps from it and close it. It factors in the overhead of the read and has a function that is used to calculate elapsed time from a start and stop timestamp.

TIMER.C This file is a sample C program that can be used to time other applications. It also uses the device driver.

To build the device driver and demonstration programs, set your current directory to `\WATCOM\SRC\OS2\PDD` and type:

```
wmake
```

To install the device driver, put the following statement in your `CONFIG.SYS` file.

```
DEVICE=\WATCOM\SRC\OS2\PDD\HRTIMER.SYS
```

You must then reboot OS/2.

To run the test program, use the following command-line:

```
HRTEST [milliseconds]
```

For `[milliseconds],` you can enter any number (e.g., 2000 which is 2 seconds).

`HRTEST.EXE` will issue a DosSleep for the amount of milliseconds specified or will use a default if no command-line parameter is given. It will get a timestamp from the device driver before and after the DosSleep and will calculate the elapsed time of that sleep and display the results. It will do this continuously until Ctrl/C or Ctrl/Break is pressed.

Keep in mind that DosSleep has a granularity of 32 milliseconds. Any discrepancy between the number of milliseconds used for the DosSleep and the elapsed time results from the timer are the fault of this granularity, not a problem with the timer. DosSleep is used solely as a convenient method of displaying the capabilities of the driver.

To run the timer program, use the following command-line:

```
TIMER program_ name [program_ args]
```

For example, to time an OS/2 Directory command, issue the following command.

Example:

```
timer cmd /c dir c:
```

Novell NLM Programming Guide

40 Creating NetWare 386 NLM Applications

Open Watcom C/C++ supports version 4.0 of the Netware 386 API. We include the following components:

header files Header files for the Netware 4.0 API are located in the `\WATCOM\NOVH` directory.

import libraries
Import libraries for the Netware 4.0 API are located in the `\WATCOM\NOVI` directory.

libraries The C/C++ libraries for Netware 4.0 is located in the `\WATCOM\LIB386` and `\WATCOM\LIB386\NETWARE` directories.

debug servers Servers for remote debugging of Netware 4.0 NLMs are located in the `\WATCOM\NLM` directory. The same directory also contains the Open Watcom Execution Sampler for NLMs.

Applications built for version 4.0 will run on 4.1. We do not include support for any API specific to version 4.1. Netware developers must use the support included with Open Watcom C/C++ version 10.0 or greater since the version supplied by Novell only works with Open Watcom C/C++ version 9.5. Netware 4.1 support requires modification to the header files supplied by Novell. Contact Novell for more information.

The following special notes apply to developing applications for NetWare.

1. You must compile your source files with the "-bt=NETWARE" option. This will cause the compiler to:

   - use the small memory model instead of the flat memory model,
   - use stack-based calling conventions,
   - search the **NETWARE_INCLUDE** environment variable before searching the **INCLUDE** environment variable, and
   - reference a special startup symbol, `__WATCOM_Prelude`, in the libraries.

2. You must compile your source files with the small memory model option ("ms"). This is accomplished by specifying the "-bt=NETWARE" option.

3. You must compile your source files with one of the stack-based calling convention options ("3s", "4s" or "5s"). This is accomplished by specifying the "-bt=NETWARE" option.

4. You must set the **NETWARE_INCLUDE** environment variable to point to the `\WATCOM\NOVH` directory. This environment variable will be searched first when you compile with the "-bt=NETWARE" option. Alternatively, you can set the **INCLUDE** environment variable to include `\WATCOM\NOVH` before other include directories.

5. If you are using the compile and link utility WCL386, you must use the following options: "-l=NETWARE -bt=NETWARE".

6. You must specify

```
system NETWARE
```

when linking an NLM. This is automatic if you are using WCL386 and the "-l=NETWARE" option.

7. If you are using other Netware APIs such as NWSNUT, then you must include `module` and `import` statements as input to the Open Watcom Linker.

   Example:

   ```
   module nwsnut
   import @%WATCOM%\novi\nwsnut.imp
   ```

   This is done automatically for the C Library (CLIB.IMP). The following import lists have been provided for Netware API libraries.

   ```
   AIO.IMP
   APPLETLK.IMP
   BSD.IMP
   CLIB.IMP
   DSAPI.IMP
   MATHLIB.IMP
   NWSNUT.IMP
   SOCKLIB.IMP
   STREAMS.IMP
   TLI.IMP
   ```

Mixed Language Programming

41 Inter-Language calls: C and FORTRAN

The purpose of this chapter is to anticipate common questions about mixed-language development using Open Watcom C/C++ and Open Watcom FORTRAN 77.

The following topics are discussed in this chapter:

- Symbol Naming Convention
- Argument Passing Convention
- Memory Model Compatibility
- Integer Type Compatibility
- How do I pass integers from C to a FORTRAN function?
- How do I pass integers from FORTRAN to a C function?
- How do I pass a string from a C function to FORTRAN?
- How do I pass a string from FORTRAN to a C function?
- How do I access a FORTRAN common block from within C?
- How do I call a C function that accepts a variable number of arguments?

41.1 Symbol Naming Convention

The symbol naming convention describes how a symbol in source form is mapped to its object form. Because of this mapping, the name generated in the object file may differ from its original source form.

Default symbol naming conventions vary between compilers. Open Watcom C/C++ prefixes an underscore character to the beginning of variable names and appends an underscore to the end of function names during the compilation process. Open Watcom FORTRAN 77 converts symbols to upper case. Auxiliary pragmas can be used to resolve this inconsistency.

Pragmas are compiler directives which can provide several capabilities, one of which is to provide information used for code generation. When calling a FORTRAN subprogram from C, we want to instruct the compiler NOT to append the underscore at the end of the function name and to convert the name to upper case. This is achieved by using the following C auxiliary pragma:

```
#pragma aux ftnname "^";
```

The "^" character tells the compiler to convert the symbol name "ftnname" to upper case; no underscore character will be appended. This solves potential linker problems with "ftnname" since (by C convention) the linker would attempt to resolve a reference to "ftnname_".

When calling C functions from FORTRAN, we need to instruct the compiler to add the underscore at the end of the function name, and to convert the name to lower case. Since the FORTRAN compiler automatically converts identifiers to uppercase, it is necessary to force the compiler to emit an equivalent lowercase name. Both of these things can be done with the following FORTRAN auxiliary pragma:

```
*$pragma aux CNAME "!_ "
```

There is another less convenient way to do this as shown in the following:

```
*$pragma aux CNAME "cname_ "
```

In the latter example, the case of the name in quotation marks is preserved.

Use of these pragmas resolves the naming differences, however, the issue of argument passing must still be resolved.

41.2 Argument Passing Convention

In general, C uses call-by-value (passes argument values) while FORTRAN uses call-by-reference (passes pointers to argument values). This implies that to pass arguments to a FORTRAN subprogram we must pass the addresses of arguments rather than their values. C uses the "&" character to signify "address of".

Example:

```
result = ftnname( &arg );
```

When calling a C function from FORTRAN, the pragma used to correct the naming conventions must also instruct the compiler that the C function is expecting values, not addresses.

```
*$pragma aux CNAME "!_ " parm (value)
```

The "parm (value)" addition instructs the FORTRAN compiler to pass values, instead of addresses.

Character data (strings) are an exception to the general case when used as arguments. In C, strings are not thought of as a whole entity, but rather as an "array of characters". Since strings are not considered scalar arguments, they are referenced differently in both C and FORTRAN. This is described in more detail in a following section.

41.3 Memory Model Compatibility

While it is really not an issue with the 32-bit compilers (both use the default "flat" memory model), it is important to know that the default memory model used in Open Watcom FORTRAN 77 applications is the "large" memory model ("ml") with "medium" and "huge" memory models as options. Since the 16-bit Open Watcom C/C++ default is the "small" memory model, you must specify the correct memory model when compiling your C/C++ code with the 16-bit C or C++ compiler.

41.4 Linking Considerations

When both C/C++ and FORTRAN object files are combined into an executable program or dynamic link library, it is important that you list a least one of the FORTRAN object files first in the Open Watcom Linker (WLINK) "FILES" directive to guarantee the proper search order of the FORTRAN and C run-time libraries. If you place a C/C++ object file first, you may inadvertently cause the wrong version of run-time initialization routines to be loaded by the linker.

41.5 Integer Type Compatibility

In general, the number of bytes used to store an integer type is implementation dependent. In FORTRAN, the default size of an integer type is always 4 bytes, while in C/C++, the size is architecture dependent. The size of an "int" is 2 bytes for the 16-bit Open Watcom C/C++ compilers and 4 bytes for the 32-bit compilers while the size of a "long" is 4 bytes regardless of architecture. It is safest to prototype the function in C, specifying exactly what size integers are being used. The byte sizes are as follows:

1. LONG - 4 bytes
2. SHORT - 2 bytes

Since FORTRAN uses a default of 4 bytes, we should specify the "long" keyword in C for integer types.

Example:

```
long int ftnname( long int *, long int *, long int * );
```

In this case, "ftnname" takes three "pointers to long ints" as arguments, and returns a "long int". By specifying that the arguments are pointers, and not values, and by specifying "long int" for the return type, this prototype has solved the problems of argument passing and integer type compatibility.

41.6 How do I pass integers from C to a FORTRAN function?

The following Open Watcom C/C++ routine passes three integers to a FORTRAN function that returns an integer value.

```
/* MIX1C.C - This C program calls a FORTRAN function to
 *           compute the max of three numbers.
 *
 * Compile/Link: wcl /ml mix1c mix1f.obj /fe=mix1
 *               wcl386  mix1c mix1f.obj /fe=mix1
 */

#include <stdio.h>

#pragma aux tmax3 "^";
long int tmax3( long int *, long int *, long int * );
```

```
void main()
{
    long int  result;
    long int  i, j, k;

    i = -1;
    j = 12;
    k = 5;
    result = tmax3( &i, &j, &k );
    printf( "Maximum is %ld\n", result );
}
```

The FORTRAN function:

```
* MIX1F.FOR - This FORTRAN function accepts three integer
*              arguments and returns their maximum.

* Compile: wfc[386] mix1f.for

        integer function tmax3( arga, argb, argc )
        integer arga, argb, argc

        tmax3 = arga
        if ( argb .gt. tmax3 ) tmax3 = argb
        if ( argc .gt. tmax3 ) tmax3 = argc
        end
```

41.7 How do I pass integers from FORTRAN to a C function?

The following Open Watcom FORTRAN 77 routine passes three integers to a Open Watcom C/C++ function that returns an integer value.

```
*  MIX2F.FOR - This FORTRAN program calls a C function to
*              compute the max of three numbers.
*
* Compile/Link: wfl[386] mix2f mix2c.obj /fe=mix2

*$pragma aux tmax3 "!_ " parm (value)

        program mix2f

        integer*4  tmax3
        integer*4  result
        integer*4  i, j, k

        i = -1
        j = 12
        k = 5
        result = tmax3( i, j, k )
        print *, 'Maximum is ', result
        end
```

The C function "tmax3" is shown below.

```
/* MIX2C.C - This C function accepts 3 integer arguments
 *           and returns their maximum.
 *
 * Compile: wcc /ml mix2c
 *          wcc386  mix2c
 */

long int tmax3( long int arga,
                long int argb,
                long int argc )
{
    long int   result;
    result = arga;
    if( argb > result ) result = argb;
    if( argc > result ) result = argc;
    return( result );
}
```

41.8 How do I pass a string from a C function to FORTRAN?

Character strings are referenced differently in C and FORTRAN. The C language terminates its strings with a null character as an End-Of-String (EOS) marker. In this case, C need not store the length of the string in memory. FORTRAN, however, does not use any EOS marker; hence it must store each string's length in memory.

The structure FORTRAN uses to keep track of character data is called a "string descriptor" which consists of a pointer to the character data (2, 4, or 6 bytes, depending on the data model) followed by an unsigned integer length (2 bytes or 4 bytes, depending on the data model).

system option	size of pointer	size of length
16-bit /MM	16 bits	16 bits
16-bit /ML	32 bits	16 bits
32-bit /MF	32 bits	32 bits
32-bit /ML	48 bits	32 bits

In order to access character data, FORTRAN needs to have access to the data's string descriptor. Hence, FORTRAN expects a pointer to a string descriptor to be passed as an argument for character data.

Passing string arguments between C and FORTRAN is a simple task of describing a struct type in C containing the two fields described above. The first field must contain the pointer to the character data, and the second field must contain the length of the string being passed. A pointer to this structure can then be passed to FORTRAN.

```
* MIX3F.FOR - This FORTRAN program calls a function written
*              in C that passes back a string.
*
* Compile/Link: wfl[386] mix3f mix3c.obj /fe=mix3

        program mix3f

        character*80 sendstr
        character*80 cstring
```

```
        cstring = sendstr()
        print *, cstring(1:lentrim(cstring))
        end
```

The C function "sendstr" is shown below.

```
/* MIX3C.C - This C function passes a string back to its
 *           calling FORTRAN program.
 *
 * Compile: wcc /ml mix3c
 *          wcc386  mix3c
 */

#include <string.h>

#pragma aux sendstr "^";

typedef struct descriptor {
    char          *addr;
    unsigned       len;
} descriptor;

void sendstr( descriptor *ftn_ str_ desc )
{
    ftn_ str_ desc->addr = "This is a C string";
    ftn_ str_ desc->len  = strlen( ftn_ str_ desc->addr );
}
```

41.9 How do I pass a string from FORTRAN to a C function?

By default, FORTRAN passes the address of the string descriptor when passing strings. If the C function knows it is being passed a string descriptor address, then it is very similar to the above example. If the C function is expecting normal C-type strings, then a FORTRAN pragma can be used to pass the string correctly. When the Open Watcom FORTRAN 77 compiler pragma to pass by value is used for strings, then just a pointer to the string is passed.

Example:

```
*$pragma aux cname "!_ " parm (value)
```

The following example FORTRAN mainline defines a string, and passes it to a C function that prints it out.

```
* MIX4F.FOR - This FORTRAN program calls a function written
*              in C and passes it a string.
*
* Compile/Link: wfl[386] mix4f mix4c.obj /fe=mix4

*$pragma aux cstr "!_ " parm (value)

        program mix4f

        character*80 forstring

        forstring = 'This is a FORTRAN string'//char(0)
        call cstr( forstring )
        end
```

The C function:

```
/* MIX4C.C - This C function prints a string passed from
 *           FORTRAN.
 *
 * Compile: wcc /ml mix4c
 *          wcc386  mix4c
 */

#include <stdio.h>

void cstr( char *instring )
{
    printf( "%s\n", instring );
}
```

41.10 How do I access a FORTRAN common block from within C?

The following code demonstrates a technique for accessing a FORTRAN common block in a C routine. The C routine defines an extern struct to correspond to the FORTRAN common block.

```
* MIX5F.FOR - This program shows how a FORTRAN common
*              block can be accessed from C.
*
* Compile/Link: wfl[386] mix5f mix5c.obj /fe=mix5

        program mix5f
        external put
        common/cblk/i,j

        i=12
        j=10
        call put
        print *, 'i = ', i
        print *, 'j = ', j
        end
```

The C function:

```
/* MIX5C.C - This code shows how to access a FORTRAN
 *           common block from C.
 *
 * Compile: wcc /ml mix5c
 *          wcc386  mix5c
 */
#include <stdio.h>

#pragma aux put "^";
#pragma aux cblk "^";
```

```
#ifdef __386__
#define FAR
#else
#define FAR far
#endif

extern struct cb {
    long int i,j;
} FAR cblk;

void put( void )
{
    printf( "i = %ld\n", cblk.i );
    printf( "j = %ld\n", cblk.j );
    cblk.i++;
    cblk.j++;
}
```

For the 16-bit C compiler, the common block "cblk" is described as `far` to force a load of the segment portion of the address. Otherwise, since the object is smaller than 32K (the default data threshold), it is assumed to be located in the DGROUP group which is accessed through the SS segment register.

41.11 How do I call a C function that accepts a variable number of arguments?

One capability that C possesses is the ability to define functions that accept variable number of arguments. This feature is not present, however, in the definition of the FORTRAN 77 language. As a result, a special pragma is required to call these kinds of functions.

```
*$pragma aux printf "!_" parm (value) caller []
```

The "caller" specifies that the caller will pop the arguments from the stack. The "[]" indicates that there are no arguments passed in registers because the `printf` function takes a variable number of arguments passed on the stack. The following example is a FORTRAN function that uses this pragma. It calls the `printf` function to print the value 47 on the screen.

```
* MIX6.FOR - This FORTRAN program calls the C
*            printf function.

* Compile/Link: wfl[386] mix6

*$pragma aux printf "!_" parm (value) caller []

        program mix6

        character cr/z0d/, nullchar/z00/

        call printf( 'Value is %ld.'//cr//nullchar, 47 )
        end
```

For more information on the pragmas that are used extensively during inter-language programming, please refer to the chapter entitled "Pragmas" in both the *Open Watcom C/C++ User's Guide* and the *Open Watcom FORTRAN 77 User's Guide.*

Common Problems

42 Commonly Asked Questions and Answers

As with any sophisticated piece of software, there are topics that are not directly addressed by the descriptive portions of the manuals. The purpose of this chapter is to anticipate common questions concerning Open Watcom C/C++. It is difficult to predict what topics will prove to be useful but with that in mind, we hope that this chapter will help our customers make full use of Open Watcom C/C++.

A number of example programs are presented throughout. The source text for these files can be found in the `\WATCOM\SAMPLES\GOODIES` directory.

The purpose of this chapter is to present some of the more commonly asked questions from our users and the answers to these questions. The following topics are discussed:

- How do I determine my current patch level?
- How do I convert to Open Watcom C/C++?
- What should I know about optimization?
- Why can't the compiler find "stdio.h"?
- How do I resolve an "Undefined Reference" linker error?
- Why aren't my variables set to zero?
- What does "size of DGROUP exceeds 64K" mean for 16-bit applications?
- What does "NULL assignment detected" mean in 16-bit applications?
- What does "Stack Overflow!" mean?
- Why do I get redefinition errors from WLINK?
- How can I open more than 20 files at a time?
- How can I see my source files in the debugger?
- What is the difference between the "d1" and "d2" compiler options?

42.1 Determining my current patch level

In an effort to immediately correct any problems discovered in the originally shipped product, Open Watcom provides patches as a continued service to its customers. To determine the current patch level of your Open Watcom software, a TECHINFO utility program has been provided. This program will display your current environment variables, the patch level of various Open Watcom software programs, and other pertinent information, such as your `AUTOEXEC.BAT` and `CONFIG.SYS` files. This information proves to be very useful when reporting a problem to the Technical Support team.

To run TECHINFO, you must ensure the Open Watcom environment variable has been set to the directory where your Open Watcom software has been installed. TECHINFO will pause after each screenful of information. The output is also placed in the file `TECHINFO.OUT`.

Below is an example of some partial output produced by running the TECHINFO utility:

Example:

```
WATCOM's Techinfo Utility, Version 1.4
Current Time: Thu Oct 27 15:58:34 1994

WATCOM                          Phone: (519) 884-0702
415 Phillip St.                 Fax: (519) 747-4971
Waterloo, Ontario
CANADA    N2L 3X2

-------------WATCOM C Environment Variables -------------
WATCOM=<c:\watcom>
EDPATH=<c:\watcom\eddat>
INCLUDE=<c:\watcom\h;c:\watcom\h\os2>
FINCLUDE=<c:\watcom\src\fortran;c:\watcom\src\fortran\win>
LIBOS2=<c:\watcom\lib286\os2;c:\watcom\lib286>
PATH=<c:\dos;c:\windows;c:\watcom\binw>
TMP=<h:\temp>
File 'c:\watcom\binw\wcc386.exe' has been patched to level '.d'
...etc...
```

In this example, the software has been patched to level "d". In most cases, all tools will share a common patch level. However, there are instances where certain tools have been patched to one level while others are patched to a different level. For example, the compiler may be patched to level "d" while the debugger is only patched to level "c". Basically, this means that there were no debugger changes in the D-level patches.

If you run the TECHINFO utility, and determine that you are not at the current patch level, it is recommended that you update your software. Patches are available on Open Watcom's bulletin board, Open Watcom's FTP site and CompuServe. They are available 24 hours a day. Patches are also available on the current release CD-ROM. Each patch will include a batch file that allows you to apply the patches to your existing software. Note that patches must be applied in sequential order, as each patch depends on the previous one.

42.2 Converting to Open Watcom C/C++

There are some common steps involved in converting C programs written for other compilers. Conversion from UNIX and other IBM-compatible PC compilers will be covered in detail later. There are six major problems with most programs that are ported to Open Watcom C/C++. The assumptions that most foreign programs make that may be invalid when using Open Watcom C/C++ are:

1. sizeof(pointer) == sizeof(int)

   (true for 16-bit systems except "far" pointers, true for 32-bit systems except "far" pointers)

2. sizeof(long) == sizeof(int)

   (not true for 16-bit systems)

3. sizeof(short) == sizeof(int)

   (not true for 32-bit systems)

4. arguments are always passed on the stack

5. dereferencing the `NULL` pointer

6. "char" is either signed or unsigned

These assumptions are very easy to make when developing programs for only one system. The first point becomes important when you move a program to 80x86 systems. Depending on the memory model, the size of an integer might not equal the size of a pointer. You might ask how this assumption is made in programs. The C language will assume that a function returns an integer unless told otherwise. If a programmer does not declare a function as returning a pointer, the compiler will generate code which would convert an integer to a pointer. On other systems, where the size of an integer is equal to the size of a pointer this would amount to nothing because no conversion was necessary (to change size). The older C compilers did not worry about warning the programmer about this condition and as such this error is imbedded in a lot of older C code. As C was moved to other machines, it became apparent that this assumption was no longer valid for all machines. The 80x86 architecture can have 16-bit integers and 32-bit pointers (in the compact, large, and huge memory models), which means that more care must be taken when working with declarations (converting an int to a 32-bit pointer will result in a segment value of 0x0000 or 0xffff). Similarly, the 386 architecture can have 32-bit integers and 48-bit pointers.

The Open Watcom C/C++ compiler will complain about incorrect pointer and integer mixing thus making programs compiled with Open Watcom C/C++ much more portable. For instance, if the Open Watcom C/C++ compiler complains about your usage of the "malloc" memory allocation function then you probably forgot to include "<stdlib.h>" which contains the prototype of the "malloc" function.

Example:

```
extern void *malloc( unsigned );
```

The Open Watcom C/C++ compiler was complaining about you trying to assign an integer (the value returned by "malloc") to a pointer. By including the header file with the correct prototype, the Open Watcom C/C++ compiler can validate that you are in fact assigning a pointer value to a pointer.

Passing arguments on the stack has been the method used by most older compilers because it allowed the C library function "printf" to work with a variable number of arguments. Older C compilers catered to a few functions by forcing all the argument handling to be handled by the caller of the function. With the advent of the ANSI (and later ISO) standard, which forced all functions expecting a variable number of arguments to be declared properly, compilers can generate smaller code for routines that did not require a variable number of arguments.

Example:

```
/* function accepting two arguments */
extern FILE *fopen( char *, char * );
/* function accepting a variable number of arguments */
extern int printf( char *, ... );
```

The Open Watcom C/C++ compiler takes advantage of this part of the ISO/ANSI standard by passing arguments in registers (for the first few arguments). If there are not enough registers for all of the arguments, the rest of the arguments are passed on the stack but the routine being called is responsible for removing them from the stack. By default, the Open Watcom C/C++ compiler uses this calling convention because it results in faster procedure calls and smaller code. The Open Watcom C/C++ calling convention carries with it a responsibility to ensure that all functions are prototyped correctly before they are used. For instance, if a procedure is called with too few arguments, the assumptions that the code generator made (while generating the code) will be invalidated. The code generator assumes that AX (EAX for the 32-bit compiler) and any other registers used to pass arguments will be modified by the called function. The code generator also assumes that the exact amount of arguments pushed on the stack will be removed by the function that is called. It is important to recognize this aspect of the Open Watcom C/C++ compiler because the program will simply not work unless the caller and the function being called strictly agree on the number and types of the arguments being passed. See the "Assembly Language Considerations" chapter in the ***Open Watcom C/C++ User's Guide*** for more details.

Some compilers allow the NULL pointer to be dereferenced and return NULL (we have never understood the rationale behind this, nor why some compilers continue to support this type of code). Leaving the aesthetics of this type of code behind, using the NULL dereferencing assumption in a program will ensure that the program will not be portable. Source code which contains the NULL dereferencing assumption must be corrected before it will work with Open Watcom C/C++.

Programs that assume that the "char" type is "signed" should use the Open Watcom C/C++ compiler "j" option. The "j" option will indicate to the Open Watcom C/C++ compiler that the "char" type is "signed" rather than the default "unsigned".

42.2.1 Conversion from UNIX compilers

The ISO/ANSI standard for C (which Open Watcom C/C++ adheres to) is very similar to UNIX C. Most of the effort in converting UNIX C programs will involve replacing references to library functions (such as the CURSES library). There are many third-party libraries which are implementations of UNIX libraries on IBM-compatible Personal Computers. There is a common problem which many older UNIX programs exhibit, namely, functions that accept a variable number of arguments are coded in many different ways. Functions accepting a variable number of arguments must be coded according to the ISO standard if they are to work with Open Watcom C/C++. We will code an example of a function which will return the maximum of a list of positive integers.

```
/*
  variable number of arguments example
*/
#include <stdarg.h>

int MaxList( int how_many, ... )
{
    va_list args;
    int max;

    max = 0;
    va_start( args, how_many );
    while( how_many > 0 ) {
        value = va_arg( args, int );
        if( value > max ) {
            max = value;
        }
    }
    va_end( args );

    return( max );
}
```

Notice that the standard header file STDARG.H must be included in any source file which defines a function that handles a variable number of arguments. The function "MaxList" must be prototyped correctly in other source files external to the source file containing the definition of "MaxList".

```
extern int MaxList( int how_many, ... );
```

See the ***Open Watcom C Library Reference*** manual description of "va_arg" for a more complete description of variable number of arguments handling.

42.2.2 Conversion from IBM-compatible PC compilers

Most of the compilers available for IBM-compatible PCs have been following the ISO/ANSI standard and, as such, the majority of programs will not require extensive source changes. There are problems with programs that use compiler-specific library functions. The use of compiler-specific library functions can be dealt with in two different ways:

1. use equivalent Open Watcom C/C++ library functions
2. write your own library functions

If portability must be maintained with the other compiler, the predefined macro **"__WATCOMC__"** can be used to conditionally compile the correct code for the Open Watcom C/C++ compiler.

The default calling convention for the Open Watcom C/C++ compiler is different from the calling convention used by other compilers for Intel-based personal computers. The Open Watcom C/C++ calling convention is different because it will pass some arguments in registers (thus reducing the overhead of a function call) rather than pushing all of the arguments on the stack. The Open Watcom C/C++ compiler is flexible enough to use different calling conventions on a per function basis. Converting code from other compilers usually involves recompiling the C source files and setting up prototypes (to use the older calling convention) for functions written in assembly language. For instance, if you have the functions "video_init", "video_put", and "video_get" written in assembly language, you can use the following prototypes in any source file which uses these functions.

```
#include <stddef.h>

extern int  cdecl video_init( void );
extern void cdecl video_put( int row,int col,char ch,int attr );
extern char cdecl video_get( int row,int col );
```

The inclusion of the `STDDEF.H` header file defines the "cdecl" calling convention. The Open Watcom C/C++ compiler will ensure that any calls to these three functions will adhere to the "cdecl" calling conventions. The Open Watcom C/C++ compiler will put a trailing underscore "_" character (as opposed to the beginning of the name for the "cdecl" convention) on any function names to ensure that the program will not link register calling convention calls to "cdecl" convention functions (or vice versa). If the linker indicates that functions defined in assembler files cannot be resolved, it could be a result of not prototyping the functions properly as "cdecl" functions.

Hint: (16-bit applications only) Most 16-bit C compilers (including Open Watcom C/C++) have a "large" memory model which means that four byte pointers are used for both code and data references. A subtle point to watch out for involves differences between memory model definitions of different compilers. The "cdecl" calling convention allows functions to assume that the DS segment register points to the group "DGROUP". The Open Watcom C/C++ large memory model has what is called a "floating DS". Any function used for the large memory model cannot assume that the DS segment register points to the group "DGROUP". There are a few possible recourses.

1. The assembly code could save and restore the DS segment register and set DS to DGROUP in order to conform to the Open Watcom C/C++ convention. If there are only a few accesses to DGROUP data, it is advisable to use the SS segment register which points to DGROUP in the large memory model.

2. The assembly function could be described using a pragma that states that DS should point to "DGROUP" before calling the function.

   #pragma aux _Setcolor __parm __loadds

   In the above example, *_Setcolor* is the sample function being described.

3. The final alternative would be the use of the "zdp" compiler option. The "zdp" option informs the code generator that the DS register must always point to "DGROUP". This is the default in the small, medium and flat memory models. Note that "flat" is a 32-bit memory model only.

42.3 What you should know about optimization

The C/C++ language contains features which allow simpler compilers to generate code of reasonable quality. Register declarations and imbedding assignments in expressions are two of the ways that C allows the programmer to "help" the compiler generate good quality code. An important point about the Open Watcom C/C++ compiler is that it is not as important (as it is with other compilers) to "help" the compiler. In order to make good decisions about code generation, the Open Watcom C/C++ compiler uses modern optimization techniques.

Hint: The definitive reference on compiler design is the "dragon" book "Compilers - Principles, Techniques, and Tools", Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, published by Addison-Wesley, Reading, Massachusetts, 1986. The authors of the "dragon" book advocate a conservative approach to code generation where optimizations must preserve the semantics of the original program. The conservative approach is used throughout the Open Watcom C/C++ compiler to ensure that programmers can use the compiler without worrying about the semantics of their program being changed. The programmer can request that potentially unsafe optimizations be performed. With regard to the "oa" (ignore aliasing) option provided by the Open Watcom C/C++ compiler, the compiler only ignores aliasing of global variables rather than ignore aliasing totally like other compilers.

There are certain pieces of information which the compiler cannot derive from the source code. The "#pragma" compiler directive is used to provide extra information to the compiler. It is necessary to have a complete understanding of both C/C++ and the machine architecture (i.e., 80x86) before using the powerful

pragma compiler directives. See the "Pragmas" chapter in the ***Open Watcom C/C++ User's Guide*** for more details.

Debugging optimized programs is difficult because variables can be assigned to different locations (i.e., memory or registers) in different parts of the function. The "d2" compiler option will restrict the amount of optimization so that variables occupy one location and can be easily displayed. It follows that the "d2" option is useful for initial development but production programs should be compiled with only the "d1" option for the best code quality. Before you distribute your application to others, you may wish to use the Open Watcom Strip Utility (WSTRIP) to remove debugging information from the executable image on disk thereby reducing disk space requirements.

> ***Hint:*** The "d2" compiler option will generate symbolic information (for every local variable) and line number information for the source file. The "d1" compiler option will only generate line number information for the source file. The use of these options determines what kind of information will be available for the particular module during the debugging session.

Incorrect programs can sometimes work when compiled with the "d2" option and not work when compiled with the "d1" option. One way this sort of problem arises involves local arrays.

```
void example( void )
{
    int i;
    int a[10];

    for( i = 0; i <= 10; ++i )
        a[i] = i;
    do_ something( a );
}
```

The "for" loop initializes one too many array elements but the version compiled with the "d2" option will overwrite the variable "i" without causing any problems. The same function compiled with the "d1" option would have the variable "i" in a register. The erroneous access of "a[10]" would modify a value that is used to restore a register when the function returns. The register would be "restored" with an incorrect value and this would affect the execution of the function that called this function. The above example shows how a program can work when compiled with the "d2" option and stop working when compiled with the "d1" option. You should always test your program fully with all the modules compiled with the "d1" option to protect yourself from any surprises.

42.4 The compiler cannot find "stdio.h"

The standard header files are usually located in the sub-directory that the Open Watcom C/C++ compiler is installed in. Suppose that the header files are located in the sub-directory `c:\watcom\h`. If the compiler indicates (through an error message) that it is unable to locate the file `STDIO.H`, we have forgotten something. There are two ways to indicate to the Open Watcom C/C++ compiler the location of the standard header files.

1. use the **INCLUDE** environment variable
2. use the "i" option (Open Watcom C/C++, Open Watcom Compile and Link)

The use of the environment variable is the simplest way to ensure that the include files will be found. For instance, if you include the following line in your system initialization file, `AUTOEXEC.BAT`,

```
set include=c:\watcom\h
```

the Open Watcom C/C++ compiler will be able to find the standard include files. The use of the "i" option is another way to give the directory name of the standard include files.

Example:

```
C>wcc myfile.c -ic:\watcom\h
    or
C>wcc386 myfile.c -ic:\watcom\h
    or
C>wpp myfile.cpp -ic:\watcom\h
    or
C>wpp386 myfile.cpp -ic:\watcom\h
```

The usual manner that these methods are combined is as follows. The **INCLUDE** environment variable is used to give the location of the standard C library header files. Any directories of header files local to a specific programming project are often candidates for the "i" option method. See the "Open Watcom C/C++ #include File Processing" section of the chapter entitled "The Open Watcom C/C++ Compilers" in the ***Open Watcom C/C++ User's Guide*** for more details.

42.5 Resolving an "Undefined Reference" linker error

The Open Watcom Linker builds an executable file by a process of resolving references to functions or data items that are declared in other source files. Certain conditions arise that cause the linker to generate an "Undefined Reference" error message. An "Undefined Reference" error message will be displayed by the linker when it cannot find a function or data item that was referenced in the program. Verify that you have included all the required object modules in the linker command and that you are linking with the correct libraries. There are a couple of "undefined references" that require some explanation.

cstart The unresolved reference for `_cstart_` indicates that the linker cannot find the C/C++ run-time libraries. The 16-bit C run-time libraries for the small memory model are `clibs.lib` and, either `maths.lib`, or `math87s.lib`. The 32-bit C run-time libraries for the flat memory model compiled for the register-based argument passing model are `clib3r.lib` and, either `math3r.lib`, or `math387r.lib`. Ensure that the **WATCOM** environment variable is set to the directory that Open Watcom C/C++ was installed in.

fltused The `_fltused_` undefined reference indicates that floating-point arithmetic has been used in the modules that exhibit this error. The remedy is to ensure that the linker can find the appropriate math library. For the 16-bit small memory model, it is either `maths.lib`, or `math87s.lib` For the 32-bit register-based argument passing model, it is either `math3r.lib`, or `math387r.lib` depending on which floating-point option is used. Ensure that the **WATCOM** environment variable is set to the directory that Open Watcom C/C++ was installed in.

_small_code_ If this undefined reference occurs when you are trying to create a 16-bit application, we have saved you many hours of debugging! The reason for this undefined reference is that the "main" entry point has been compiled for a big code model (in any one of medium, large, or huge memory models). Any of the modules that have this undefined reference have been compiled for a small code model (in any one of small or compact memory models) and as such do not have the correct return instructions. You should recompile the modules so that all the modules are compiled for the same memory model. Combining

source modules compiled for different memory models is very difficult and often leads to strange bugs. If your program has special considerations and this reference causes you problems, there is a "work-around". You could resolve the reference with a PUBLIC declaration in an assembler file or code the following in Open Watcom C/C++.

```
/* rest of your module */

void _small_code( void )
{}
```

The code generator will generate a single RET instruction with the public symbol `_small_code_` attached to it. The common epilogue optimizations will probably combine this function with another function's RET instruction and you will not even pay the small penalty of one byte of extra code.

There may be another cause of this problem, the "main" function must be entered in lower case letters ("Main" or "MAIN" are not identified as being the same as "main" by the compiler). The compiler will identify the module that contains the definition of the function "main" by creating the public definition of either `_small_code_` or `_big_code_` depending on the memory model it was compiled in.

_big_code_ Your module that contains the "main" entry point has been compiled with a 16-bit small code model (small or compact). The modules that have this undefined reference have been compiled in 16-bit big code models (medium, large, or huge). You should recompile the modules so that all the modules are compiled in the same memory model. See the explanation for `_small_code_` for more details.

main_ All C programs, except applications developed specifically for Microsoft Windows, must have a function called "main". The name "main" must be in lower case for the compiler to generate the appropriate information in the "main" module.

WINMAIN All Windows programs must have a function called "WinMain". The function "WinMain" must be declared "pascal" in order that the compiler generate the appropriate name in the "WinMain" module.

42.6 Why my variables are not set to zero

The linker is the program that handles the organization of code and data and builds the executable file. C guarantees that all global and static uninitialized data will contain zeros. The "BSS" region contains all uninitialized global and static data for C programs (the name "BSS" is a remnant of the early UNIX C compilers). Most C compilers take advantage of this situation by not explicitly storing all the zeros to achieve smaller executable file sizes. In order for the program to work correctly, there must be some code (that will be executed before "main") that will clear the "BSS" region. The code that is executed before "main" is called "startup" code. The linker must indicate to the startup code where the "BSS" region is located. In order to do this, the Open Watcom Linker (WLINK) treats the "BSS" segment (region) in a special manner. The special variables '_edata' and '_end' are constructed by the Open Watcom Linker so that the startup code knows the beginning and end of the "BSS" region.

Some users may prefer to use the linker provided by another compiler vendor for development. In order to have the program execute correctly, some extra care must be taken with other linkers. For instance, with the Microsoft linker (LINK) you must ensure that the '/DOSSEG' command line option is used. With the Phar Lap Linker, you must use the "-DOSORDER" command line option. In general, if you must use other linkers, extract the module that contains `_cstart` from `clib?.lib` (? will change depending on the

memory model) and specify the object file containing `_cstart` as the first object file to be processed by the linker. The object file will contain the information necessary for the linker to build the executable file correctly.

42.7 What does "size of DGROUP exceeds 64K" mean for 16-bit applications?

This question applies to 16-bit applications. There are two types of segments in which data is stored. The two types of segments are classified as "near" and "far". There is only one "near" segment while there may be many "far" segments. The single "near" segment is provided for quick access to data but is limited to less than 64K in size. Conversely, the "far" segments can hold more than 64K of data but suffer from a slight execution time penalty for accessing the data. The "near" segment is linked by arranging for the different parts of the "near" segment to fall into a group called DGROUP. See the section entitled "Memory Layout" in the ***Open Watcom Linker User's Guide*** for more details.

The 8086 architecture cannot support segments larger than 64K. As a result, if the size of DGROUP exceeds 64K, the program cannot execute correctly. The basic idea behind solving this problem is to move data out of the single "near" segment into one or more "far" segments. Of course, this solution does not come without any penalties. The penalty is paid in decreased execution speed as a result of accessing "far" data items. The magnitude of this execution speed penalty depends on the behavior of the program and, as such, cannot be predicted (i.e., we cannot say that the program will take precisely 5% longer to execute). The specific solution to this problem depends on the memory model being used in the compilation of the program.

If you are compiling with the tiny, small, or medium memory models then there are two possible solutions. The first solution involves changing the program source code so that any large data items are declared as "far" data items and accessed with "far" pointers. The addition of the "far" keyword into the source code makes the source code non-portable but this might be an acceptable tradeoff. See the "Advanced Types" chapter in the ***Open Watcom C Language Reference*** manual for details on the use of the "near" and "far" keywords. The second solution is to change memory models and use the large or compact memory model. The use of the large or compact memory model allows the compiler to use "far" segments to store data items that are larger than 32K.

The large and compact memory models will only allocate data items into "far" segments if the size of the data item exceeds 32K. If the size of DGROUP exceeds 64K then a good solution is to reduce the size threshold so that smaller data items will be stored into "far" segments. The relevant compiler option to accomplish this task is "zt<num>". The "zt" option sets a data size threshold which, if exceeded, will allocate the data item in "far" segments. For instance, if the option "zt100" is used, any data item larger than 100 bytes will be allocated in "far" segments. A good starting value for the data threshold is 32 bytes (i.e., "zt32"). The number of compilations necessary to reduce the size of DGROUP for a successful link with WLINK depends on the program. Minimally, any files which allocate a lot of data items should be recompiled. The "zt<num>" option should be used for all subsequent compiles, but the recompilation of all the source files in the program is not necessary. If the "DGROUP exceeds 64K" WLINK error persists, the threshold used in the "zt<num>" option should be reduced and all of the source files should be recompiled.

42.8 What does "NULL assignment detected" mean in 16-bit applications?

This question applies to 16-bit applications. The C language makes use of the concept of a `NULL` pointer. The `NULL` pointer cannot be dereferenced according to the ISO standard. The Open Watcom C/C++ compiler cannot signal the programmer when the `NULL` address has been written to or read from because the Intel-based personal computers do not have the necessary hardware support. The best that the run-time system can do is help programmers find these sorts of errors through indirect means. The lower 32 bytes of "near" memory have been seeded with 32 bytes of the value 0x01. The C run-time function "_exit" checks these 32 bytes to ensure that they have not been written over. Any modification of these 32 bytes results in the "NULL assignment error" being printed before the program terminates.

Here is an overview of a good debugging technique for this sort of error:

1. use the Open Watcom Debugger to debug the program
2. let the program execute
3. find out what memory has been incorrectly modified
4. set a watchpoint on the modified memory address
5. restart the program with the watchpoint active
6. let the program execute, for a second time
7. when the memory location is modified, execution will be suspended

We will go through the commands that are executed for this debugging session. First of all, we invoke the Open Watcom Debugger from the command line as follows:

```
C>wd myprog
```

Once we are in the debugger type:

```
DBG>go
```

The program will now execute to completion. At this point we can look at the output screen with the debugger command, "FLIP".

```
DBG>flip
```

We would see that the program had the run-time error "NULL assignment detected". At this point, all we have to do is find out what memory locations were modified by the program.

The following command will display the lower 16 bytes of "near" memory.

```
DBG>examine _ _ nullarea
```

The command should display 16 bytes of value 0x01. Press the space bar to display the next 16 bytes of memory. This should also display 16 bytes of value 0x01. Notice that the following data has two bytes which have been erroneously modified by the program.

```
_ _ nullarea     01 01 56 12 01 01 01 01-01 01 01 01 01 01 01 01
_ _ nullarea+16  01 01 01 01 01 01 01 01-01 01 01 01 01 01 01 01
```

The idea behind this debugging technique is to set a watchpoint on the modified memory so that execution of the program will be suspended when it modifies the memory. The following command will "watch" the memory for you.

```
DBG>watch __ nullarea+2
```

There has to be a way to restart the program without leaving the Open Watcom Debugger so that the watchpoint is active during a subsequent execution of the program. The Open Watcom Debugger command "NEW" will reload the program and prepare for a new invocation of the program.

```
DBG>new
DBG>go
```

The Open Watcom Debugger command "GO" will start execution of the program. You may notice that the program executes much slower than usual but eventually the debugger will show the part of the program that modified the two bytes. At this point, you might want to clear the watchpoint and proceed to debug why the memory was modified. The command to clear the watchpoint is:

```
DBG>watch/clear 1
```

The "1" indicates that you want watchpoint number 1 to be cleared. Typing "WATCH" by itself will print out all active watchpoints. The above technique is generally useful for any type of memory overwrite error provided you know which memory location has been overwritten.

> ***Hint:*** The Open Watcom Debugger allows many commands to have short forms. For instance, the "EXAMINE" command can be shortened to an "E". We used the full commands in the examples for clarity.

42.9 What "Stack Overflow!" means

The memory used for local variables is allocated from the function call stack although the Open Watcom compilers will often use registers for local variables. The size of the function call stack is limited at link-time and it is possible to exceed the amount of stack space during execution. The Open Watcom run-time library will perform checks whenever a large amount of stack space is required by a function but it is up to the user to check stack requirements before calling a Open Watcom run-time function. Compiling programs with stack checking will ensure that there is enough stack space to call a Open Watcom run-time function.

There are various ways of protecting against stack overflow errors. First, one should minimize the number of recursive functions used in an application program. This can be done by recoding recursive functions to use loops. Keep the amount of stack used in functions to a minimum by using and reusing static arrays whenever possible. These techniques will reduce the amount of stack space required but there still may be times where the default amount of stack space is insufficient. The Open Watcom Linker (WLINK) allows the user to set the amount of stack space at link-time through the directive "OPTION STACK=size" where size may be specified in bytes with an optional "k" suffix for kilobytes (1024 bytes).

Example:
```
option stack=9k
```

Debugging a program that reports a stack overflow error can be accomplished with the following sequence.

1. Load your application into the debugger
2. Set a breakpoint at `__STKOVERFLOW`
3. Run the application until the breakpoint at `__STKOVERFLOW` is triggered
4. Issue the debugger "show calls" command. This will display a stack traceback giving you the path of calls that led up to the stack overflow situation.

The solution to the stack overflow problem at this point depends on the programmer.

42.10 Why redefinition errors are issued from WLINK

This question comes up often in discussions about porting old UNIX or Microsoft C programs. The problem stems from the forgiving nature of early UNIX linkers. In early C code, it was common to define header files like this:

Example:
```
/* define global variables */
int line_count;
int word_count;
int char_count;
```

The header file would then be included in many different modules. The C compiler would generate a definition of each variable in each module and leave it to the linker to pick one and resolve all references to one variable. The development of the ANSI C standard made this practice non-conforming. The Open Watcom C compiler is an ISO/ANSI C compiler and as such, is not required to support this obsolete behavior. The effect is that WLINK will report redefinition errors. The header file must be coded in such a way that the variables are defined in one module. One way to do this is as follows:

Example:
```
#ifdef DEFINE_HERE
#define GLOBAL
#else
#define GLOBAL  extern
#endif
/* define global variables */
GLOBAL int line_count;
GLOBAL int word_count;
GLOBAL int char_count;
```

In most modules, the macro "DEFINE_HERE" will not be defined so the file will be equivalent to:

Example:

```
/* define global variables */
extern int line_ count;
extern int word_ count;
extern int char_ count;
```

In one module, the macro "DEFINE_HERE" must be defined before the header file is included. This can be done by defining the macro on the command line or by coding like this:

Example:

```
#define DEFINE_ HERE
#include "globals.h"
```

42.11 How more than 20 files at a time can be opened

The number of file handles allowed by Open Watcom C/C++ is initialized to 20 in `stdio.h`, but this can be changed by the application developer. To change the number of file handles allowed with Open Watcom C/C++, follow the steps outlined below.

1. Let `n` represent the number of files the application developer wishes to have open. Ensure that the *stdin, stdout, stderr, stdaux,* and *stdprn* files are included in the count.

2. Change the `CONFIG.SYS` file to include "files=n" where "n" is the number of file handles required by the application plus an additional 5 handles for the standard files (this applies to DOS 5.0). The number "n" may vary depending on your operating system and version. If you are running a network such as Novell's NetWare, this will also affect the number of available file handles. In this case, you may have to increase the number specified in the "files=n" statement.

3. Add a call to `_ grow_ handles` in your application.

The following example illustrates the use of `_ grow_ handles.`

Example:

```
/*
 *  HANDLES.C
 *  This C program grows the number of file handles so
 *  more than 16 files can be opened. This program
 *  illustrates the interaction between _ grow_ handles and
 *  the DOS 5.0 file system. If you are running a network
 *  such as Novell's NetWare, this will also affect the
 *  number of available file handles. In the actual trial,
 *  FILES=40 was specified in CONFIG.SYS.
 */
```

```
#include <stdio.h>

void main()
{
    int    i, j, maxh, maxo;
    FILE  *temp_ files[50];

    for( i = 25; i < 40; i++ ) {
        /* count 5 for stdin,stdout,stderr,stdaux,stdprn */
        printf( "Trying for %2.2d handles...", 5 + i );
        maxh = _ grow_ handles( 5 + i );
        maxo = 0;
        for( j = 0; j < maxh; j++ ) {
            temp_ files[j] = tmpfile();
            if( temp_ files[j] == NULL )break;
            maxo++;
        }
        printf( " %d/%d temp files opened\n", maxo, maxh );
        for( j = 0; j < maxo; j++ ) {
            fclose( temp_ files[j] );
        }
    }
}
```

42.12 How source files can be seen in the debugger

The selection and use of debugging information is important for getting the most out of the Open Watcom Debugger. If you are not able to see your source code in the Open Watcom Debugger source window, there are three areas where things may have gone wrong, namely:

1. using the correct option for the Open Watcom C/C++.
2. using the correct directives for the Open Watcom Linker.
3. using the right commands in the Open Watcom Debugger.

The Open Watcom C/C++ compiler takes C/C++ source and creates an object file containing the generated code. By default, no debugging information is included in the object file. The compiler will output debugging information into the object file if you specify a debugging option during the compile. There are two levels of debugging information that the compiler can generate:

1. Line numbers and local variables ("d2" option)
2. Line numbers ("d1" option)

The options are used to determine how much debugging information will be visible when you are debugging a particular module. If you use the "d2" option, you will be able to see your source file and display your local variables. The "d1" option will display the source but will not give you access to local variable information.

The Open Watcom Linker (WLINK) is the tool that puts together a complete program and sets up the debugging information for all the modules in the executable file. There is a linker directive that indicates to the linker when it should include debugging information from the modules. There are five levels of debugging information that can be collected during the link. These are:

1. global names (DEBUG)

2. global names, line numbers (DEBUG LINE)
3. global names, types (DEBUG TYPES)
4. global names, local variables (DEBUG LOCALS)
5. all of the above (DEBUG ALL)

Notice that global names will always be included in any request for debugging information. The debugging options can be combined

```
DEBUG LINE, TYPES
```

with the above directive resulting in full line number and typing information being available during debugging. The directives are position dependent so you must precede any object files and libraries with the debugging directive. For instance, if the file `mylink.lnk` contained:

```
#
# invoke with: wlink @mylink
#
file main
debug line
file input, output
debug all
file process
```

then the modules `input` and `output` will have global names and source line information available during debugging. All debugging information in the module `process` will be available during debugging.

> ***Hint:*** A subtle point to debugging information is that all the modules will have global names available if any debugging directive is used. In the above example, the module `main` will have global name information even though it does not have a DEBUG directive preceding it.

It is preferable to have one DEBUG directive before any FILE and LIBRARY directives. You might wonder if this increases the size of the executable file so that it will occupy too much memory during debugging. The debugging information is loaded "on demand" by the debugger during the debugging session. A small amount of memory (40k default, selectable with the Open Watcom Debugger "dynamic" command line option) is used to hold the most recently used module debugging information. In practice, this approach saves a lot of memory because most debugging information is never used. The overhead of accessing the disk for debugging information is negligible compared to accessing the source file information. In other words, you can have as much debugging information as you want included in the executable file without sacrificing memory required by the program. See the section entitled "The DEBUG Directive" in the ***Open Watcom Linker User's Guide*** for more details.

If the previous steps have been followed, you should be well on your way to debugging your programs with source line information. There are instances where the Open Watcom Debugger cannot find the appropriate source file even though it knows all the line numbers. The problem that has surfaced involves how the source file is associated with the debugging information of the module. The original location of the source file is included in the debugging information for a module. The name that is included in the debugging information is the original name that was on the Open Watcom C/C++ command line. If the original filename is no longer valid (i.e., you have moved the executable to another directory), the Open Watcom Debugger must be told where to find the source files. The Open Watcom Debugger "Source Path" menu item (under "File") can be used to supply new directories to search for source files. If your source files are located in two directories, the following paths can be added in the Open Watcom Debugger:

```
c:\program\c\*.c
c:\program\new\c\*.c
```

The "*" character indicates where the module name will be inserted while the Open Watcom Debugger is searching for the source file. See the description of the "Source Path" menu item in the ***Open Watcom Debugger User's Guide*** for more details.

42.13 The difference between the "d1" and "d2" compiler options

The reason that there are two levels of debugging information available is that the code optimizer can perform many more optimizations and still maintain "d1" (line) information. The "d2" option forces the code optimizer to ensure that any local variable can be displayed at any time in the function. To illustrate why this results in less optimum code being generated for a function, let us look at a simple array initialization.

```
extern int a[100];

void init_ a( void )
{
    int i;

    for( i = 0; i < 100; ++i ) {
        a[i] = 3*i;
    }
}
```

The code optimizer will ensure that you can print the value of the variable "i" at any time during the execution of the loop. The "d2" option will always generate code and debugging information so that you can print the value of any variable during the execution of the function. In order to get the best code possible and still see your source file while debugging, the "d1" option only generates line number information into the object file. With line number information, much better code can be generated. Here is the C equivalent of the code generated for the array initialization example.

```
extern int a[100];

void init_ a( void )
{
    int *t1;
    int t2;

    /* for( i = 0; i < 100; ++i ) { */
    t1 = a;
    t2 = 0;
    do {
        /* a[i] = 3*i; */
        *t1 = t2;
        ++t1;
        t2 += 3;
    /* } */
    } while( t1 != a + 100 );
}
```

The above code executes very quickly but notice that the variable "i" has been split into two different variables. One of the variables handles the use of "i" as an array index and the other handles the calculation of "3*i". The debugging of programs that have undergone extensive optimization can be difficult, but with the source line information it is much easier. To summarize, use the "d2" compiler option if you are developing a module and you would like to be able to display each local variable. The "d1" compiler option will give you line number information and the best generated code possible. There is absolutely no reason not to specify the "d1" option because the code quality will be identical to code generated without the "d1" option.

1

3

4

8

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X