Introduction to OS/2 Programming

OS/2 Awareness Series
Volume II

One UP Corporation

Introduction to OS/2

Programming

OS/2 Awareness Series, Volume 2

One Up Corporation

Printed in U.S.A.
on recycled paper, can be recycled.

Introduction to OS/2

Programming

ISBN 1-884988-01-6

Acknowledgements

The authors of this document are:

Feite Kraay
One Up Computer Services Ltd., Toronto, Canada

Larrry Pollis
One Up Corporation, Dallas, Texas

Craig Chambers
One Up Corporation, Dallas, Texas

Technical validation was done by:

Dan Kardell
One Up Corporation, Dallas, Texas

Richard Dews
One Up Corporation, Dallas, Texas

Page Design was done by:

Jon Morey
One Up Corporation

Cover design was done by:

Larry Pollis
One Up Corporation, Dallas, Texas

The project leader and editor for this project was:

Craig Chambers
One Up Corporation, Dallas, Texas

Special thanks go to Libby Boyd, of Dove Oaks Publishing.

FOREWORD

We are in the midst of a major revolution. The 90s will be known as the start of the digital age. It is almost impossible today to pick up a magazine or newspaper without seeing at least one story about the digital revolution as it gains momentum. If a modern day Rip Van Winkle had fallen asleep when the Apollo 11 crew returned from the moon and were to awaken today, he would not recognize the world.

When he went to sleep, a desktop "computer" was a mechanical calculating machine. He used a slide rule in his engineering classes. He had an 8-track tape player in his car and listened to LP records for entertainment at home. He made his home movies on 8mm film. Today, he would have to look hard to find any of these items in the stores. At home and in our cars, we listen to our music on digital CD players. We record music on Digital Compact Cassettes or digital Mini-Disks. Our desktop calculator is now a high speed workstation with 8-16 MB RAM. Virtual Reality is becoming a reality.

The computer industry is bringing all of these technologies together on your desktop. They call it Multimedia. You can now listen to digital CD quality music, watch full motion TV, calculate a spreadsheet, and maintain active communications with several remote systems, all while typing a letter with a word processing program.

The keystone of this technology is the operating system. For desktop systems, this key component is called OS/2. The current version, 2.1, is the most advanced operating environment available today for integrating these technologies. A truly revolutionary system, OS/2 is the first system designed for the new digital world.

There is a second revolution sweeping the computer industry. When Rip went to sleep, most interactive data was entered from keyboards. Today, most applications are designed with a graphical user interface or GUI. Introduced in 1984 with Apple's Lisa system, the GUI concept has put a new face on the computer user interface. OS/2 is the most advanced expression of this concept. Its Workplace

Shell makes the system look and behave like the user's familiar desktop.

The purpose of the OS/2 Awareness series is to help you understand how OS/2 enables you to participate in the digital and GUI revolutions. This volume, the second in the series, provides an introduction to OS/2 programming. It shows you how to create programs that take advantage of the powerful multitasking capabilities built into OS/2. It also shows you how to use the Presentation Manager to give your program a graphical user interface.

Volume 1 in this series provides an overview of the OS/2 system itself. It describes how you can move from the older, limited function PC-DOS and Windows environment to a fully capable OS/2 environment. Other volumes in this series cover topics such as how to enable OS/2 to communicate effectively with remote systems, and how to integrate database capabilities into OS/2.

This book is a collaboration between IBM Corporation's Southwestern Area staff and One Up Corporation's staff. The authors have been working with OS/2 since its earliest design phases in the mid-1980s. In this book they use their perspective and experience to explain how to begin writing programs for OS/2 version 2.1 that fully exploit its powerful new capabilities.

I hope that you find this book both informative and enjoyable.

Craig Chambers

Table of Contents

Introduction

- Welcome
- Overview
- About This Book
- Summary
- Directions
- Copyright Information

Welcome

Welcome to the *One Up OS/2 Awareness Series*. This series is a set of books that are designed to help you become familiar with the capabilities of IBM's OS/2 2.1 operating system. It is assumed that you have a basic familiarity with IBM compatible computers and PC-DOS.

Overview

This volume, the second in the series, will increase your understanding of the OS/2 development environment. Hopefully, after reading this book, you will appreciate why OS/2 is considered the best workstation-based development platform available today.

Whether you are developing applications for DOS-based systems, Windows-based systems, or native OS/2 applications, the "crash protection" that is built into the OS/2 system makes OS/2 the development platform of choice. OS/2 is a powerful 32-bit preemptive multitasking operating system. That means that everything happens faster than with other systems because OS/2 takes full advantage of the capabilities of the 32-bit system processor.

Its multitasking capabilities allow you to run multiple programs simultaneously, each in its own protected area of memory. This allows you to have your editor, debugger, and program being tested all active at the same time in separate sessions. You can quickly and easily move between them as you develop your application.

OS/2 programs can be written more efficiently than for DOS or Windows because the complications of extended, expanded, and segmented memory are eliminated by OS/2's flat memory system. With its High Performance File System (HPFS), OS/2 also eliminates the restriction of file names to just eleven characters.

About This Book

In the first chapter, we describe the OS/2 programming environment. The IBM OS/2 Developer's Toolkit 2.1 is introduced and each component is described briefly. This is followed by a description of the IBM C/C++ Tools 2.0 compiler and SourceLink, a very powerful editing tool from One Up Corporation. The chapter ends with a brief discussion of the application build procedure.

We then get down to the business of actually writing an OS/2 program. The base OS/2 API is introduced, including discussions about multi-threading, memory management, and the OS/2 file management functions. You will write a program that uses all of these system functions.

Then a Presentation Manager, or GUI, interface is added to the program developed earlier. This interface includes a menu bar and scroll bars. The chapter opens with a discussion about the concept of object oriented programming and what that means is real life applications. The fundamentals of Presentation Manager architecture are then covered, which introduces the PM API. By the end of the chapter, these concepts have been incorporated into a working PM program that also includes multi-threading and memory management concepts.

Summary

This book introduces OS/2 and Presentation Manager programming concepts and describes the OS/2 development environment. Hopefully, you will be able to take the sample programs and use them as the basis for your own OS/2 programs.

This is just an introduction however. It will be necessary to get a deeper knowledge of the system if you are to develop professional quality applications. There are several excellent books available that go into great detail about how to write OS/2 programs. Also, there are excellent OS/2 and Presentation Manager Programming classes available from One Up Corporation.

Directions

There are other volumes in the *One Up OS/2 Awareness Series*. Volume 1 is an overview of the OS/2 2.1 system. It includes a description of the features and capabilities of OS/2. It also includes an overview of the installation process and offers some troubleshooting guidance.

Volume 3 discusses OS/2 communications support. It includes discussions of the of the OS/2 LAN Server system, Communications Manager/2, and LAN Adapter Protocol Support (LAPS). These books include installation and configuration information, as well as product descriptions and usage hints and tips.

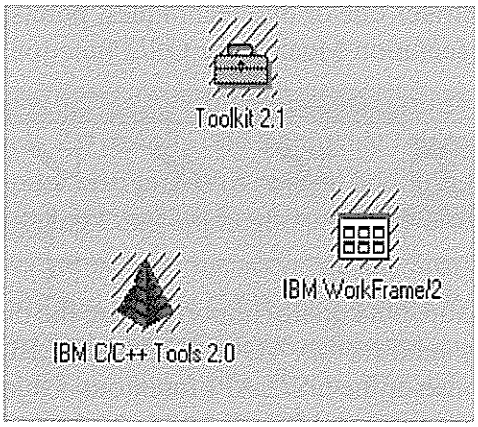
Copyright Information

The OS/2 Programming Environment

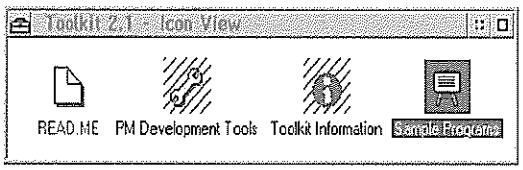
- The IBM OS/2 Developer's Toolkit 2.1
- The IBM C/C++ Tools 2.0
- The Application Build Procedure

The IBM OS/2 Developer's Toolkit 2.1

Development in OS/2, like most environments, requires at least two things: a language compiler supported by the environment, and libraries of API (Application Programming Interface) functions that allow the programmer to take full advantage of the features of that environment. This publication will focus on the use of the IBM C/C++ Tools 2.0 and the IBM OS/2 Developer's Toolkit 2.1, for building 32-bit OS/2 2.0 or OS/2 2.1 applications. Also available is the IBM Developer's WorkFrame/2; a customizable, graphical workbench environment to facilitate maintenance of development projects.

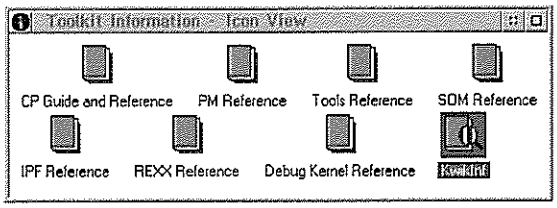


The toolkit supplies several components of the development process. When installed, these can be accessed through folders on the desktop.

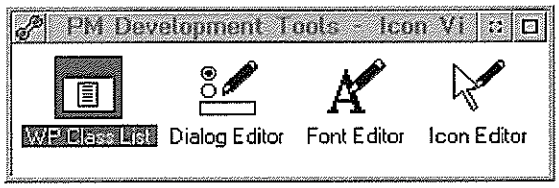


The Toolkit Information folder provides on-line reference manuals for OS/2 development. Primary among these are the *PM Reference* and *CP Reference*, which list function syntax and data type definitions for all of the Presentation Manager and base OS/2 API functions, respectively. It is very worthwhile to become familiar with the table of contents, index, and search capabilities of these manuals, as they are the first line of assistance in coding OS/2 APIs. The API functions are classified by their use, and this is indicated by the function names. For example, all calls beginning with the prefix Dos... are calls to OS/2 base function, and can be found in the CP Reference under subheadings such as Memory Management, Execution Control, etc. All functions beginning with Win... call the PM Window Manager, and all functions beginning with Gpi... call the Graphics Programming Interface. These, and many other categories, are documented in the PM Reference. Index files are installed along with these

reference manuals, which are accessible to the OS/2 enhanced editor. When editing a source file, a keystroke combination enables the user to immediately find the on line reference for a specific API function or C library function.



Among the development tools installed are the icon editor and the dialog editor. These are especially useful in defining additional resources to be added to a PM application. Also, a large set of sample programs is provided with the toolkit, each of which illustrates one or two techniques, points of style, or general areas of interest in Presentation Manager or OS/2 functions.



Not visible on the desktop, but equally important, are the directories into which the toolkit installs. The subdirectory TOOLKT21\OS2LIB contains the link libraries necessary for use of the OS/2 API functions. Object files must be linked to OS2386.LIB to resolve API calls made in the application code. This library actually contains references to Dynamic Link Library files installed on every OS/2 user's machine. Libraries such as PMWIN.DLL or PMGPI.DLL contain the actual function definitions for the APIs, shared by all programs at run-time.

```
 Directory of D:\TOOLKT21

.            <DIR>       9-07-93   11:36p
..           <DIR>       9-07-93   11:36p
C            <DIR>       9-07-93   11:36p
SC           <DIR>       9-07-93   11:37p
REXX         <DIR>       9-07-93   11:37p
ASM          <DIR>       9-07-93   11:37p
CPLUS        <DIR>       9-07-93   11:37p
ICON         <DIR>       9-07-93   11:37p
BOOK         <DIR>       9-07-93   11:37p
OS2HELP      <DIR>       9-07-93   11:38p
IPFC         <DIR>       9-07-93   11:38p
READ     ME      37851   5-06-93    9:47p
DLL          <DIR>       9-07-93   11:40p
OS2BIN       <DIR>       9-07-93   11:40p
OS2LIB       <DIR>       9-07-93   11:55p
BOOKCAT  ASC     49973   5-06-93    9:47p
       16 file(s)      87824 bytes used
                     4562944 bytes free
```

The subdirectory TOOLKT21\C\OS2H contains header files with the function prototypes for all API calls, as well as all necessary data structure and data type definitions required. Selections from these header files must be included in all source files. This is accomplished by pre-processor definitions indicating which API functions are used, followed by a #include statement referencing the file OS2.H. Pre-processor tests within the header files then result in including only the sections necessary for the application.

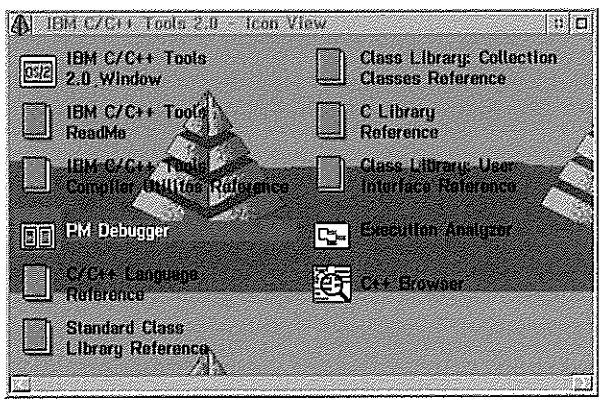
For example, the lines

```
#define INCL_WIN

#include <os2.h>
```

would direct the pre-processor to include only those sections of the header files necessary to support PM Window Management API calls. This will improve compile speed as only a minimal set of header information is read. For every API function, the on-line reference manual shows the necessary #define statement for support of that call.

The IBM C/C++ Tools 2.0

The IBM C/C++ Tools 2.0 installs into a separate folder on the desktop. This full C and C++ compiler also provides several additional utilities to assist in developing and debugging OS/2 applications. Many of the utilities apply to the C++ environment and are beyond the scope of this document. For C development, the important utilities are the *C/C++ Language Reference*, the *C Library Reference*, and the PM Debugger. The two reference manuals provide a syntax guide for the C language and library functions. The debugger, known as IPMD.EXE, or the Interactive PM Debugger, is a source-level debugger that runs in a Presentation Manager window. This provides a view of source code, by thread, as the application executes. The programmer may set break points, view register contents, and view or alter variable contents during execution.



C/C++ Tools 2.0 installs into several subdirectories on the drive. The subdirectory IBMCPP\BIN contains the compiler/linker, ICC.EXE. IBMCPP\INCLUDE contains all the C standard header files. IBMCPP\LIB contains the C language libraries, automatically picked up by the linker. There are libraries for multi-threaded and single-threaded application support, as well as static and dynamic linking to the C library functions.

```
 Directory of D:\ibmcpp

.            <DIR>      7-02-93   2:15p
..           <DIR>      7-02-93   2:15p
HELP         <DIR>      7-02-93   2:15p
DLL          <DIR>      7-02-93   2:15p
BIN          <DIR>      7-02-93   2:15p
WKFRAME      <DIR>      7-02-93   2:17p
TUTORIAL     <DIR>      7-02-93   2:17p
LOCALE       <DIR>      7-02-93   2:20p
INCLUDE      <DIR>      7-02-93   2:20p
LIB          <DIR>      7-02-93   2:23p
SAMPLES      <DIR>      7-02-93   2:23p
IBMCLASS     <DIR>      7-02-93   2:29p
SYS          <DIR>      7-02-93   2:32p
TMP          <DIR>      7-02-93   2:43p
       14 file(s)              0 bytes used
                         4560896 bytes free
```

The compiler is invoked simply by calling ICC.EXE. Printed and on-line reference manuals document all the compile switches and their meanings. Five are used in the example programs provided with this document, as follows:

```
icc /c /Ss /Ti+ /Kb /Gm browse1.c
```

/c instructs the compiler to compile only. /Ss enables support for // comments in C. /Ti+ embeds trace information into the object file to enable use of the IPMD debugger. /Kb enables additional diagnostic messages to be displayed by the compiler. /Gm compiles for the multi-threaded C libraries, which can provide better performance for many standard C functions.

Either ICC.EXE or LINK386.EXE (supplied in the TOOLKT21\OS2BIN subdirectory) may be used to link the object files. For example,

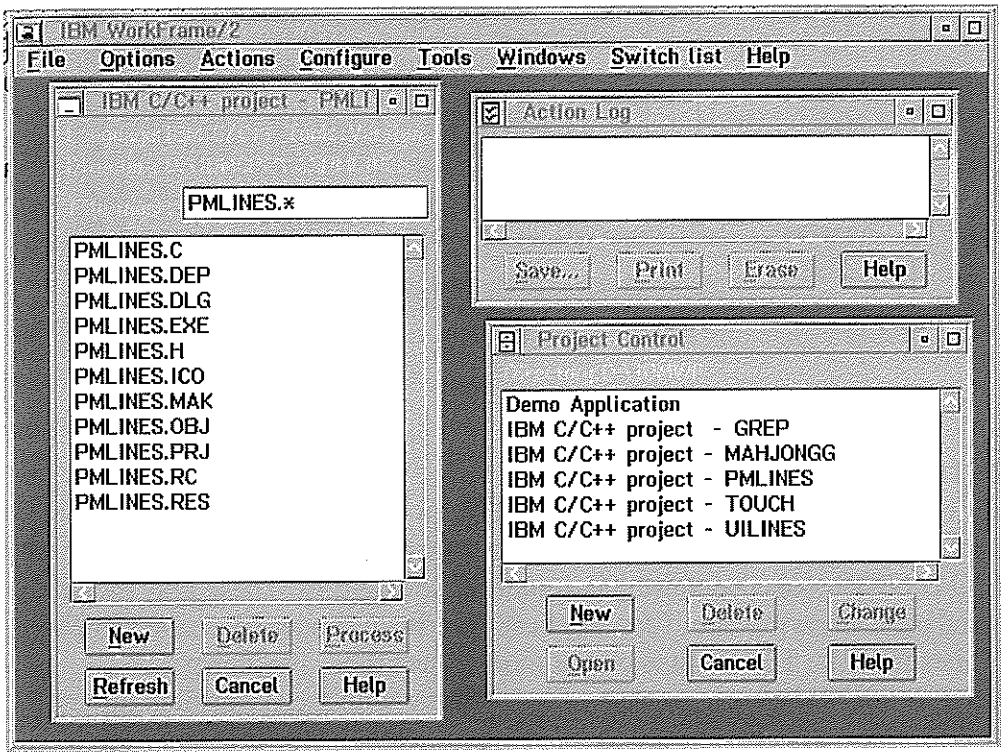
```
link386 /NOI /PM:VIO /DEBUG browse1.obj;
```

links the object file produced by the previous compile step. /NOI instructs the linker not to ignore case sensitivity in function names. /PM:... indicates the application type desired. /PM:PM would request a Presentation Manager graphical program, while /PM:VIO requests a textual command prompt application. /DEBUG again embeds support for the IPMD debugger in the executable code.

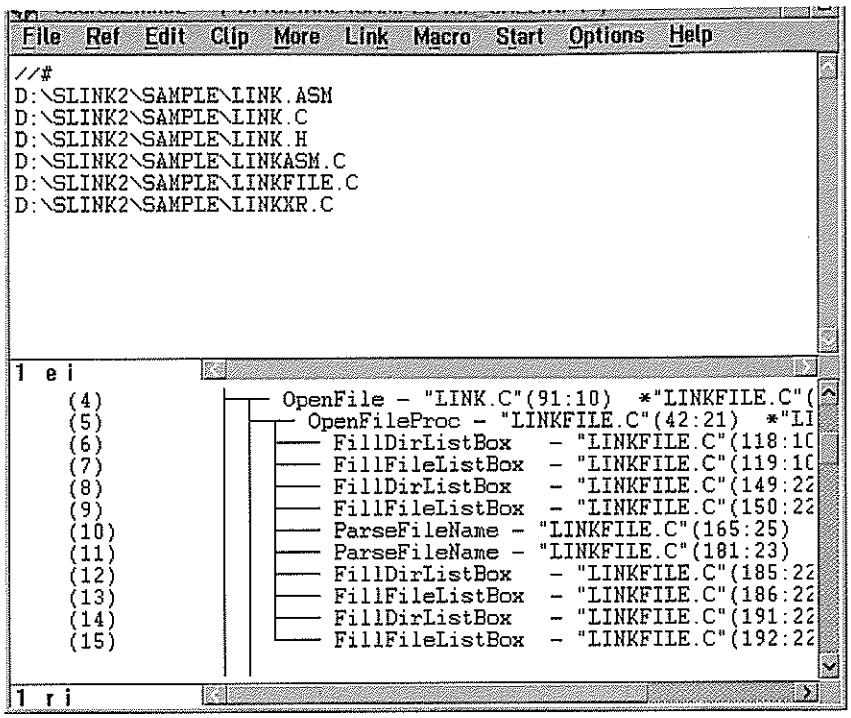
In order to use these tools, some modifications need to be made to the environment. These changes can be done automatically in the CONFIG.SYS file as the tools install. The path environment variable must refer to the BIN subdirectories in both the toolkit and the C Set, and the include environment variable must refer to the appropriate header subdirectories. The statements setting variables for help, bookshelf, progref, pmref, and helpndx are all necessary to enable the on-line help and documentation for the toolkit and C Set.

```
LIBPATH=D:\IBMWF\DLL;D:\TOOLKT21\DLL;D:\IBMCPP\DLL; ...
SET PATH=D:\IBMWF\BIN;D:\TOOLKT21\OS2BIN;D:\IBMCPP\BIN; ...
SET LIB=D:\TOOLKT21\OS2LIB;D:\IBMCPP\LIB;
SET INCLUDE=D:\TOOLKT21\C\OS2H;D:\IBMCPP\INCLUDE;D:\IBMCPP\IBMCLASS; ...
SET DPATH=D:\IBMCPP\LOCALE;D:\IBMCPP\HELP;D:\IBMCPP\SYS;D:\TOOLKT21\BOOK; ...
SET HELP=D:\IBMWF\HELP;D:\TOOLKT21\OS2HELP;D:\IBMCPP\HELP; ...
SET BOOKSHELF=D:\IBMWF\HELP;D:\TOOLKT21\BOOK;D:\IBMCPP\HELP;
SET PMREF=PMFUN.INF+PMGPI.INF+PMHOK.INF+PMMSG.INF+PMREL.INF+PMWIN.INF+PMWKP.INF
SET HELPNDX=EPMKWHLP.NDX
SET IPFC=D:\TOOLKT21\IPFC;
SET TMP=D:\IBMCPP\TMP
SET PROGREF21=CPGREF1.INF+CPGREF2.INF+CPGREF3.INF
```

IBM WorkFrame/2 is a separate product that may be installed to assist in the development process. This is a graphical, windowed environment that organizes development activity by project. A project simply refers to a subdirectory containing the code and make file for a particular program or DLL. Each project may be configured with a different set of compile and link options, which are set via dialog boxes with on-line help describing the switches. When all options are set, WorkFrame/2 can optionally build a make file based on the files specified, commands requested, and options set. WorkFrame/2 also permits the creation of composite projects, built from several base projects to facilitate management of more complex applications.



WorkFrame/2 is a modular product, and may be configured to support a variety of compilers, editors, and tools necessary to your environment. One editor tool that integrates into the WorkFrame/2 environment is SourceLink from One Up Corporation. SourceLink is a hypertext editor that greatly facilitates the management of large applications with multiple source and header files. In addition to its full editor capability, SourceLink provides a hypertext connection between source files. A call tree is constructed for all functions in the application, and via mouse clicks the user may navigate through the code, viewing and changing all instances of function calls, variables, or defined constants through all source files. SourceLink also connects to the on-line reference manuals through index files, so that help for any API function can be accessed via a single keystroke



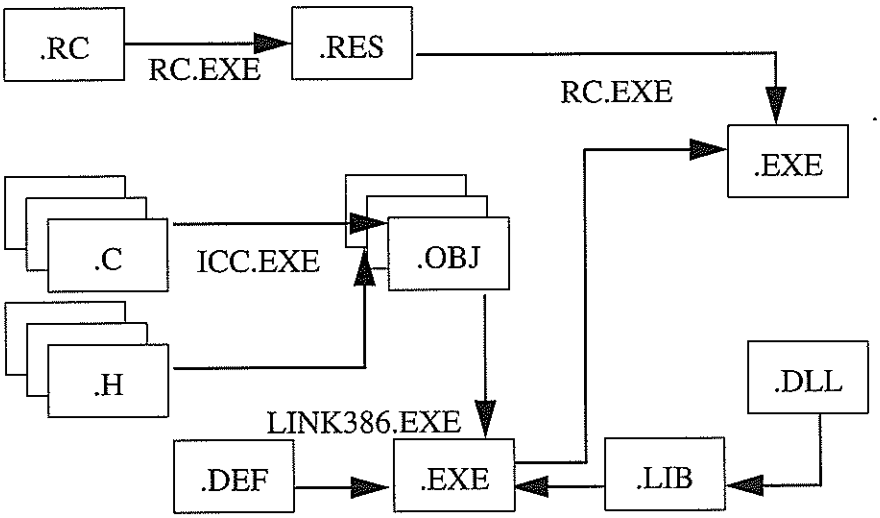
The Application Build Procedure

A number of steps are involved in building an OS/2 Presentation Manager application. First, the compile step is fairly straightforward. The compiler, ICC.EXE is invoked against one or more C source files. These may include standard C library header files, application private header files, and OS/2 toolkit header files. The result is one or more object files. The linker, LINK386.EXE links those object files together with the C run-time library as well as OS2386.LIB. This provides references to the API functions defined in the OS/2 DLLs. If the application will be referencing code in other user-defined DLLs, then usually the linker must be supplied a library file for those functions as well.

Optionally, a module definition file (.DEF) may be provided to the linker. This file can be used instead of command-line switches to indicate stack size, application type, and other features of the final executable program. It is also commonly used in the construction of DLLs, to indicate which functions are to be exported (made externally available from the DLL).

The linker, especially in the Presentation Manager environment, only produces an intermediate executable. Missing are the resources, which in PM may be thought of as the language-specific components of the application. Menus, title and message strings, dialogs, and icons are all examples of resources defined through the resource source file (.RC). The resource compiler, RC.EXE, is

invoked against the resource source file to produce an intermediate compiled version of the resource (.RES). RC.EXE is invoked again to embed the .RES file into the .EXE, in order to make the resources available to the application.



OS/2 Base API Programming

- Overview
- Multi-threading
- Memory Management
- OS/2 File Functions

Overview

Several things must come together to ensure the success of an OS/2 application. Naturally, its use of the Presentation Manager interface will greatly enhance the application's usability. This will be covered in a later section. Equally important, however, is the application's intelligent use of base operating system features to improve performance and throughput. For an OS/2 application to achieve its full potential, it should be multi-threaded and use the 32-bit linear memory model.

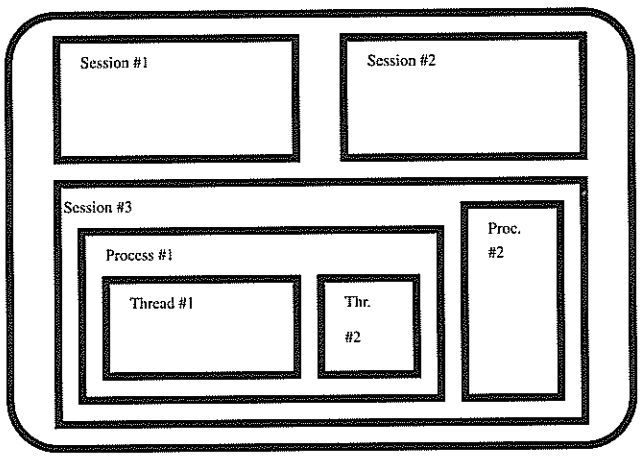
The intelligent use of threads is probably the single largest boost in performance and throughput in the OS/2 environment. A thread in an OS/2 application can be thought of as simply an asynchronous function call. In other words, instead of executing sequentially through one function call after another, an OS/2 application can invoke a thread function and return immediately to the caller, so that both functions execute simultaneously. Thus, while saving the current contents of a document to disk in a secondary thread, an editor may continue to receive and process further keystrokes from the user. Or, a spreadsheet application might do a recalculation in a secondary thread, allowing the user to continue to manipulate other cells or load and use another sheet at the same time.

The 32-bit linear memory scheme in OS/2 2.x removes the difficulty of dealing with memory segments as in earlier versions of the operating system. Instead of allocating memory as a collection of variable-sized segments, with a maximum segment size of 64K, memory objects are allocated from a flat address space with a theoretical maximum size of 4 gigabytes. Each object, no matter how big, can be treated as contiguous. Pointer manipulation becomes much easier since there is no need for concern about crossing segment boundaries. The objects are divided by the operating system into pages of 4K in size, which form the basic unit for virtual memory management. OS/2 will swap individual pages (rather than entire segments) to disk and back as necessary improving the performance of the memory manager.

The example program BROWSE1.EXE illustrates these concepts as well as the file system APIs. Source code is provided in the files BROWSE1.H and BROWSE1.C. This program reads a text file from the disk (the file name can be supplied as a command line argument) and prints the contents of the file to stdout. Reading the file is be done in a secondary thread, so that the main() function is free to perform other work, if necessary. The file is read into a dynamically allocated memory buffer. When the thread finishes, main() prints the contents of the file.

Multi-threading

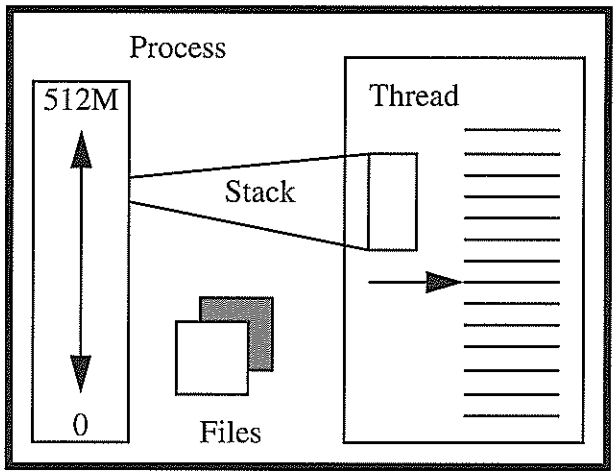
OS/2 enables multitasking at three levels. First, and most obvious to the end user, is the fact that multiple sessions can be executed on the desktop. A session may be thought of as a collection of logical devices — logical screen, keyboard and mouse, which are mapped to the physical devices if the session is in the foreground. OS/2 devotes a different session to each type of application that can be run — DOS, Windows, OS/2 Full Screen, and Presentation Manager. In other words, when a program starts up, OS/2 determines and builds the appropriate session in which to execute that program. This information is determined from the .EXE file itself, and may have been embedded by the linker.



A process, in OS/2, is an instance of an executable file loaded into memory. If a program spawns a child process (loads and executes another executable file, using the DosExecPgm() API function) then both processes are sharing the same session. That is, they are sharing the same logical (and possibly physical) devices. If two full-screen processes were to run in the same session, and write to stdout at the same time, then the output from both would appear, probably mixed, on the screen. This is the second level of multitasking available in OS/2. Normally, only one of the processes would handle screen and keyboard I/O, while other processes manage secondary tasks. A spreadsheet, for example, might start up a child process to do communication work on its behalf, downloading information from a host database to be provided to the spreadsheet via shared memory.

The Presentation Manager is known as an extended session, for it is able to share the keyboard, mouse and screen among all PM applications running. Input and output are handled on a per-window basis among all Presentation Manager programs, thus, there is no conflict or contention among PM applications for the physical devices as could happen in the full-screen environment.

Finally, the third level of multitasking is the creation of separate threads within a process. A thread is simply a function that is called asynchronously so that it executes simultaneously with its caller. All potentially long tasks (disk I/O, calculations, database search) should be executed in threads, enabling the application to continue to service user input. In fact, the rule of thumb for Presentation Manager applications is that any task taking longer than 1/10 of a second should be threaded. This avoids the annoyance of the user being "locked out," watching the clock pointer until the application responds. (Some manuals are more generous, and allow a 1/2 second rule.)

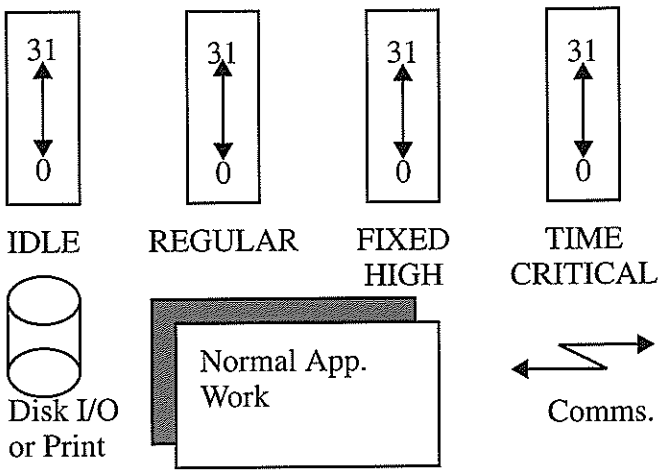


Several resources are allocated and maintained by the system for an application, some at the process level and some at the thread level. A process can be thought of as consisting of one or more threads, all sharing the same address space (theoretically 4 gigabytes, but initially limited to 512 megabytes for the sake of 16-bit compatibility) and other resources such as file handles or semaphores. These resources are allocated and protected at the level of the process, so it is the programmer's responsibility to ensure that no "collisions" might occur among several threads trying to write to the same memory location at the same time. A protection violation will only occur if a thread tries to use a resource that is not allocated to the process, for example, referencing a pointer that does not point to a valid location within the process address space. If that happens, OS/2 will terminate the process causing the violation, so that all other processes may continue to run unharmed.

The thread has its own register set and execution stack allocated to it by the operating system. When the thread is created, the stack size may be specified. It should be a minimum of 8K, and could easily be much larger with no impact on performance, since OS/2 will dynamically commit pages of real memory to the stack as required.

Every thread also has an execution priority that will determine how often it will get serviced by the CPU. Priority is assigned on a class and level basis. The four classes - Idle, Regular, Fixed High and Time Critical - are each subdivided into 32 discrete levels. A thread's priority may easily be changed using

the DosSetPriority() API call. It should seldom be necessary, however, as OS/2 will manage each thread's execution with high efficiency. A thread performing intensive work, such as print formatting, or file I/O, may be lowered to the Idle class. A thread doing communication work might need to be raised to Time Critical, to ensure that it is able to read all incoming data without interruption.



The basic principle of the scheduler is that the highest priority ready thread (i.e. a thread that could be doing work, if only it had CPU time) is allocated the CPU. Time Critical threads will preempt any other threads running. Idle class threads will only get time if the CPU is idle, when all higher priority threads are either blocked or finished. Most threads in a process will default to the Regular priority class unless they are explicitly changed. If there are more than one thread of equal, highest priority in the system, OS/2 will allocate CPU time to each as fairly as possible.

OS/2 will schedule all threads, and adjust their priorities, based on parameters set in the CONFIG.SYS file. THREADS=512 limits the system to a total of 512 threads across all processes, although this limit can be increased to a maximum of 4095. PRIORITY=DYNAMIC permits the operating system to adjust thread priorities to improve overall throughput. One boost involves use of the Fixed High priority class. When an application is brought to the foreground, the regular class threads of that foreground process are moved to the Fixed High class. The thread performing I/O (Reading the keyboard, or handling a Presentation Manager message queue) will receive a further I/O boost. The threads move back to Regular when the application is switched to the background. MAXWAIT=3 configures the starvation boost that OS/2 may apply to threads. If a thread in the Regular or Fixed High class is ready to run, and has not had access to the CPU for 3 seconds, it is boosted in priority to a level just below Time Critical, to try to ensure that it will receive the next time slice. (Of course, if a Time Critical thread is running, the Regular thread will still be preempted.) When the boosted thread's time slice is up, OS/2 moves the thread back to its original class and level. The line TIMESLICE=x,y is not normally entered into CONFIG.SYS. This forces minimum and maximum lengths for

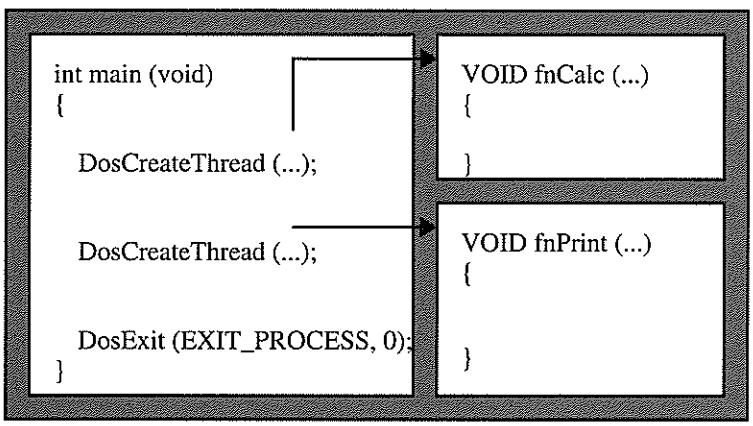
the time slices allocated to threads, but OS/2 can generally achieve better performance by dynamically setting time slice lengths based on current system load.

CONFIG.SYS

```
THREADS=512
PRIORITY=DYNAMIC
MAXWAIT=3
REM TIMESLICE=32,248
```

Every OS/2 process has at least one thread, known as the primary thread and identified as Thread One. The function main(), which is the entry point of a C program, also represents this primary thread. Other threads, known as secondary threads, are created using the DosCreateThread() API function. Each of these threads is also identified with a numeric thread ID value that is returned from DosCreateThread().

When main() terminates, the primary thread is ended. When the primary thread ends, the process is ended, regardless of the execution status of any secondary threads. A secondary thread should be notified (perhaps by a semaphore, or a message in Presentation Manager) when the process is about to terminate. The primary thread could then use the DosWaitThread() API function to ensure that the secondary thread has performed any cleanup work necessary before the process ends. DosWaitThread() allows any thread to pause or block until any other thread in the same process terminates. The API function DosExit() with the flag EXIT_PROCESS is normally used to terminate main().



DosCreateThread() is the OS/2 API function used to create a secondary thread. A zero return code from this function indicates successful completion; non-zero return codes indicate errors that are documented in the *CP Reference*. The first parameter to DosCreateThread() is the address of a variable of data type TID, or Thread ID. The thread ID is just an integer that uniquely identifies the thread being created within the process. Other API functions, such as DosKillThread() or DosWaitThread(), require the thread ID as a parameter.

The second parameter to DosCreateThread() is the name of the function which is to become the thread. This parameter is of data type PFNTHREAD, which simply means a function that uses the _System calling convention. The thread function must be prototyped and defined using _System. This is a calling convention unique to the C/C++ compiler, which identifies functions that will be called by the operating system. Threads and Window Procedures in Presentation Manager must be defined as _System, since in each case OS/2 will actually call the function on behalf of the program.

Third, DosCreateThread() accepts an unsigned long integer which will be passed as an argument to the thread function. A function that is to be a thread must therefore be prototyped and defined to accept only a single four-byte parameter. Normally, the calling function will allocate and initialize a data structure to contain all the information the thread function will need. A pointer to the data structure is then passed as a parameter to the thread, and either the caller or the thread may free the structure when the thread terminates.

BROWSE1.H

```
typedef struct   // Thread argument structure
   {
     CHAR     szFileName[255];  // File name
     HEV      hevLoad;          // Semaphore handle
     PCH      pchFile;          // Pointer to buffer
     ULONG    ulSize;           // Size of file
} OPENINFO, *POPENINFO;

// Thread function prototype
VOID _System fnOpenThread (POPENINFO pOpenInfo);
```

BROWSE1.C

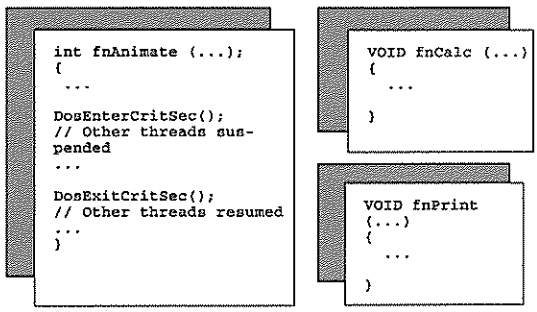
```
POPENINFO pOpenInfo;
TID       tidOpen;

pOpenInfo = (POPENINFO) malloc (sizeof (OPENINFO));

strcpy (pOpenInfo -> szFileName, argv[1]);
pOpenInfo -> hevLoad = hevLoad;
DosCreateThread (&tidOpen, (PFNTHREAD) fnOpenThread,
                 (ULONG) pOpenInfo, 0, 8192);
```

In its fourth parameter, DosCreateThread() accepts a flag that may be used to indicate the initial execution status of the thread. If bit one of the flag is 0, the thread starts executing immediately upon creation. If bit one of the flag is 1, the thread is created in the suspended state. The DosResumeThread() API would be used, later, to start the thread executing. This way, the overhead of thread creation (allocation of the stack and control block) can be dealt with at an earlier point in the code, so that when the thread is needed it can be executed with a minimum delay. Bit two of the flag affects the management of the thread's stack.

The last parameter to DosCreateThread() is used to request the size of the thread's execution stack. This size is expressed in bytes, and should be a multiple of 4096. All memory objects are rounded up in size to the nearest page. A minimum stack size of 8K is recommended, and it may often be worthwhile to request a much larger stack. As long as bit two of the flag in the fourth parameter is 0, the operating system will dynamically commit pages of real memory

to the thread's stack as needed, so that the initial overhead of a very large stack is no greater than that of a smaller stack, with the benefit that the extra space is available, if necessary, as the thread executes. If bit two of the flag is set to 1, then all the memory for the stack is committed when the thread is created.

After creating the thread, DosCreateThread() returns immediately to the calling function. Now both the caller and the thread may continue to execute, and OS/2 will assign CPU time slices to each thread in turn.



Several other API functions are available to assist in thread management. Of these, DosEnterCritSec() and DosExitCritSec() deserve particular mention. In many cases, a thread in a process will need to complete a particular set of instructions without interruption. A typical example is a thread that is dynamically updating a window's appearance in Presentation Manager, perhaps to do some simple animation work. If this thread were preempted while drawing to the window, the animation may pause or appear incomplete. Such a job is called a critical section, and the thread should call the DosEnterCritSec() API before beginning the work. This will cause all other threads in the same process to suspend, so that the animation thread is less likely to be preempted. When the critical work is complete, the thread must call DosExitCritSec() to re-enable execution of the other threads in the same process. These calls have no effect on other processes, so a thread may still be preempted by a higher priority process becoming unblocked.

DosKillThread() may be used to cause immediate termination of another thread, but it is usually safer to allow the thread to terminate itself using the DosExit() API, with the flag EXIT_THREAD. The DosExitList() API is used to register exit list functions to OS/2 on behalf of the current process. Then, when the process terminates, OS/2 keeps one thread of the process alive long enough to execute all the exit list functions. These functions could be used for process-wide clean up of resources such as memory or file handles.

By using multiple threads, an OS/2 application can significantly enhance its throughput. For example, the sample program browse1 uses the DosCreateThread() API to create a secondary thread, which will open a file and read it into memory. While the thread is busy, main() is free to do other work. In this case, it simply prints messages to the screen.

To know when the thread has finished, browse1 uses an OS/2-provided flag known as an Event Semaphore. The semaphore is created using the DosCreateEventSem() API function, and identified by a handle which is passed as part of the argument data structure to the thread. A semaphore, in general, is simply a four-byte flag in memory that has two states (on or off). The meaning of the semaphore state depends on the application context. Two base types of semaphores are provided with OS/2. Mutex (mutual exclusion) semaphores are used to serialize the execution of threads if they are all making use of the same protected resource. Event semaphores are used for synchronizing one or more threads on the occurrence of a specific event, in other words, signalling other threads that an event has occurred.

BROWSE1.C - Main Thread

```
HEV hevLoad;
...

DosCreateEventSem(NULL,     // No Name
                  &hevLoad, // Handle
                  0,        // No Flags
                  FALSE     // Initially Set
...
// Do other work... then wait for
// file to be loaded
DosWaitEventSem(hevLoad,  // Handle
                -1);      // Indefinite wait
```

BROWSE1.C - Secondary Thread

```
// Load File
...

// Notify when finished
DosPostEventSem(hevLoad);
```

The two states of the event semaphore are referred to as Set and Posted. The event semaphore behaves very much like a traffic light, with the Set state corresponding to a red light and the Posted state corresponding to green. If one or more drivers encounter a red light, they all stop. If one or more threads call DosWaitEventSem() and specify a semaphore that is in the Set state, then all those threads are blocked from further execution. When the traffic light turns to green, all waiting cars may proceed. When another thread posts the event semaphore using DosPostEventSem(), all waiting threads are unblocked and dispatched again for execution.

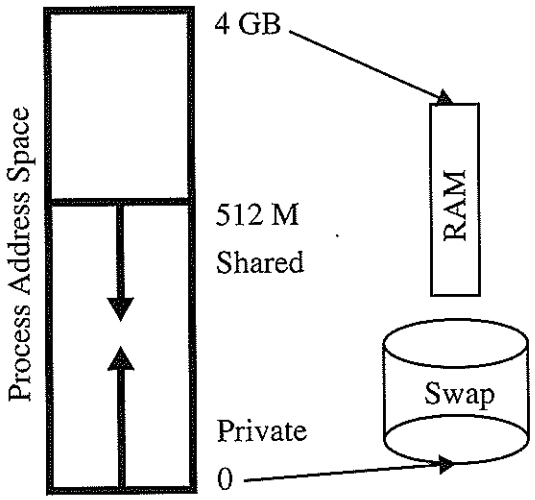
In the sample program browse1, main() creates an event semaphore that is already in the set (red light) state. The semaphore handle is passed to the secondary thread, along with the file name entered by the user, in the argument data structure. When main() has finished doing its other work it calls DosWaitEventSem() to block until the file has been completely loaded by the secondary thread. The thread calls DosPostEventSem() when it is finished. Then main() continues execution, and displays the contents of the file on the screen. Thus the event semaphore is used by the secondary thread to signal to main() that the file load operation has completed.

Memory Management

With the introduction of the 32-bit flat memory model in version 2.0, OS/2 gained a significant increase in performance. Prior versions of OS/2, based on the 16-bit Intel 80286 architecture, managed memory in segments. Each segment was variable in size, up to a maximum of 64K (the maximum that can be addressed in 16 bits). OS/2 implemented its virtual memory scheme by swapping individual segments to the SWAPPER.DAT file on disk as necessary to make room for new memory requests being made by applications. As segments were moved from RAM to disk and back, fragmentation would occur in RAM due to the variable sizes of the segments. OS/2 would then dynamically move segments in RAM to try to maximize the amount of free space available. This segment motion went on continuously, and degraded performance. Also, the fact that memory could only be allocated in units of 64K caused some programming difficulty. A file to be read into memory, for example, would have to be divided across multiple segments. Any manipulation of that file in memory, or pointer arithmetic, had to take into account the segment boundaries.

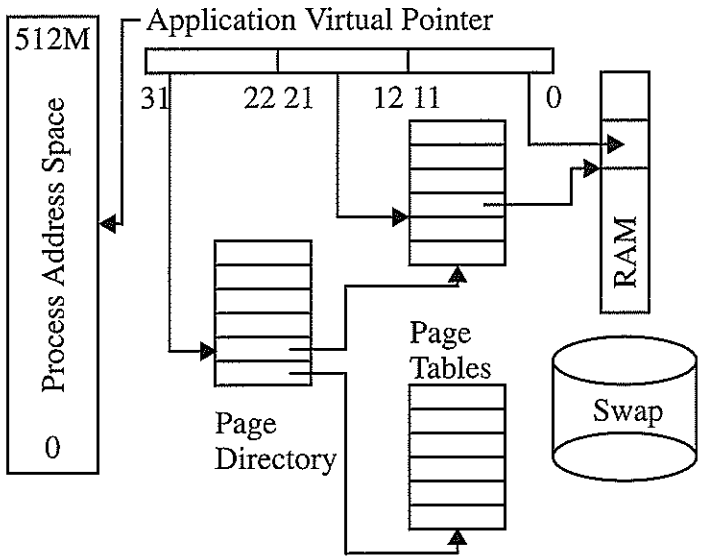
These issues all disappear with OS/2 2.0 based on the Intel 80386 architecture. This 32-bit processor still uses a segmented scheme, but segments may be up to 4 gigabytes in size. This is large enough that the operating system and all application code and data can coexist in a single segment. Memory within this segment is now divided into pages of 4K in size, which is the basis of the virtual memory implementation, almost exactly as is done in the mainframe S/370 environment. Since individual pages are swapped back and forth between RAM and SWAPPER.DAT on disk, there is no problem of fragmentation and no overhead of segment motion. Also, each process can allocate memory objects of any size. The size of an object is automatically rounded up to the nearest multiple of 4K, so that it will occupy a whole number of pages. This greatly eases the manipulation and management of large memory objects, and also improves overall operating system performance.

Although the 32-bit addressing scheme allows a theoretical virtual memory limit of 4 gigabytes, OS/2 today restricts each process to an address space of 512 megabytes in size.This provides compatibility with older 16-bit code. In the 80286 architecture, a process would access its memory segments via entries in a local descriptor table, containing 8192 entries, mapping a total of 512 megabytes. A 16-bit process running on OS/2 2.1 can still share memory with a 32-bit process on the same system. Or, a 32-bit process can call a function in a 16-bit DLL installed on the same system and share pointers with the DLL function.



As a process executes in OS/2, any private memory that it allocates (the .EXE itself, stack, or other allocations) grows up from the bottom of the address space. Any shared memory allocated or used (.DLL code, or memory allocated for the clipboard or a DDE conversation) grows down from the top of the process address space. OS/2 guarantees a 64 megabyte minimum size for both the private and shared memory regions.

Multiple processes run simultaneously on OS/2, each with its own 512 megabyte address space. Each of these must fit into limited system memory (RAM plus swap space on disk) with full protection. This is accomplished by using virtual pointers, which reference pages of memory by means of a page directory and page tables allocated to the process. Each process has exactly one page directory, which is simply a list of 1024 entries. The high-order 10 bits of a pointer (bits 31 through 22) are treated as an index into the page directory. Each page directory entry points to a page table, which is another list of 1024 entries. The middle 10 bits of the pointer (21 through 12) index into the specific page table. The page table entry points to the physical page, either in memory or swapped to disk. Finally, the lowest 12 bits of the pointer (11 through 0) provide the page offset, or the location of this pointer within the 4K of the page itself.



Since each process has a separate page directory, all pointers are therefore distinct per process and memory is protected. Since there are 1024 entries in the page directory, and 1024 entries in each page table, and each page is 4096 bytes in size, this scheme does indeed allow a 4 gigabyte address space per process, although it is currently artificially limited to 512 megabytes. As far as the process is concerned, it can be allocating and accessing memory objects that are treated as contiguous within its address space, since the page table entries pointing to the pages that comprise any object are sequential. However, those pages may be scattered in real memory as OS/2 swaps pages to disk and back. This is transparent to the application; pointer arithmetic on a memory object will simply change the page table index as a page boundary is crossed, and references are then made to the new page wherever it may exist in real memory. This scheme maximizes flexibility and performance for the operating system while also maximizing ease of use for the programmer.

The first API function for memory management is DosAllocMem(). This API is used to allocate a memory object of any size, and returns a pointer in the first parameter. The size is specified in the second parameter, and is rounded up to the nearest multiple of 4096. The last parameter to DosAllocMem() is a flag indicating attributes to be set on the memory object. DosAllocMem() returns zero to indicate successful allocation, and non-zero values indicating possible error conditions (out of memory, or invalid attributes).

The attribute flags specify the access permission to the memory as well as other characteristics of the object. These flags can be combined using the C bitwise OR (|) operator. The attributes may be changed in some cases with the DosSetMem() API call. Access permission can be one of PAG_WRITE, PAG_READ, or PAG_EXECUTE. Every object should be allocated initially with PAG_WRITE. Specifying a lower permission will deny even the process

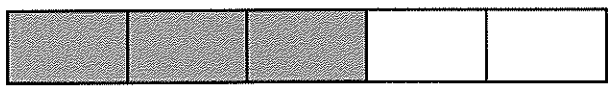
allocating the object write access to the pages, and the permission may not be increased with the DosSetMem() API function. PAG_READ or PAG_EXECUTE would really only be used in giving shared memory access to another process, to ensure that only one process may write to the memory.

```
PVOID  p;
ULONG  ulFlags, ulSize, rc;

ulFlags = PAG_WRITE;
ulSize  = 0x4000;

rc = DosAllocMem(&p,       // Pointer returned
                 ulSize,   // 16K of Memory
                 ulFlags);// Writable, uncommitted
```

One other important attribute flag is PAG_COMMIT. If PAG_COMMIT is specified when allocating a memory object, OS/2 finds physical pages of RAM to back the number of page table entries reserved for the object. In other words, the object actually occupies real memory, and is immediately usable. If, however, PAG_COMMIT is not specified, then only the page table entries are reserved, and no real memory is occupied. Such an object is called a sparse memory object. Of course, the pages must be committed (using DosSetMem() and specifying PAG_COMMIT) before they may be used. This is useful when allocating very large memory objects. If a large object is allocated sparsely, there is no initial overhead incurred by the operating system to find all the pages. As the application's use of the object grows over time, individual pages or sets of pages may be committed when needed. This way, the overhead is only incurred when absolutely necessary, and overall performance may be improved.

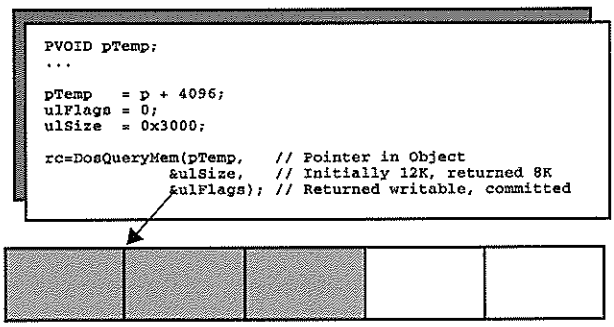
Other attribute flags are also available for memory objects. OBJ_GIVEABLE and OBJ_GETTABLE, for example, would be used with the allocation of shared memory. PAG_GUARD is used by the operating system in stack management, and may also be used to make sparse memory management a little easier and more flexible. PAG_WRITE, PAG_READ, and PAG_COMMIT, however are the most commonly specified with DosAllocMem() and DosSetMem().

```
ulFlags = PAG_WRITE | PAG_COMMIT;
ulSize  = 0x3000;

rc = DosSetMem (p,          // Pointer in Object
                ulSize,     // 12K to Set
                ulFlags);   // Writable, Committed
```

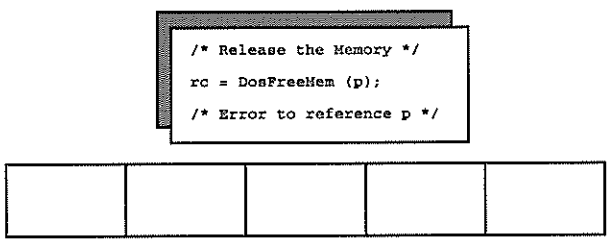


The DosSetMem() API allows an application to change the attribute flags of any range of pages in a memory object. If the access permission is to be changed, the page must be de-committed first. (Either the object was initially

allocated as sparse memory, or a range of pages could be explicitly de-committed using DosSetMem() with the PAG_DECOMMIT flag.) DosSetMem() accepts a pointer to the base of any page as its first parameter, and the size of a range of pages (expressed in bytes) as the second parameter. The flags to be set to the range of pages are passed in the third parameter. In the example above, DosSetMem() is used to commit the first two pages of a memory object, with the PAG_WRITE access permission.

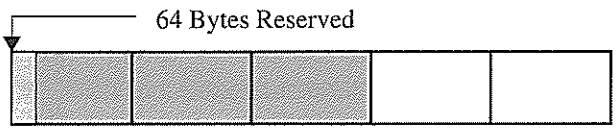


Using DosQueryMem(), an application may query the attribute flags set for any range of pages in a memory object. A pointer to the beginning of the range is passed in the first parameter, and the size of the range (in bytes) is passed in the second parameter. The flags are retrieved in the third parameter. Both the second and third parameters to this function are addresses of integers, indicating a call-by-reference. The value of the third parameter (flags) must be zero when DosQueryMem() is called, and it is set to be the current state of the flags upon return of the call. The size initially specifies the range to be queried. Upon return, it provides the size of the contiguous block of pages within the range queried that has identical flags set for all pages. For example, assume that a pointer, pTemp, points to the first of three pages within some memory object. The first two pages are writable and committed, but the third page is writable and sparse. The integer ulSize is initialized to be the size 12K, and the integer ulFlags is initialized to 0. Calling DosQueryMem(pTemp, &ulSize, &ulFlags) would return a value of 8K in ulSize, and PAG_WRITE | PAG_COMMIT in ulFlags.



Finally, DosFreeMem() is used to release an area of memory previously allocated with DosAllocMem(). DosFreeMem() accepts a single parameter, a pointer to the base of the memory object being released. When this call completes, the page table entries for the memory object (as well as the pages of real memory) are marked as free, and may be re-used by subsequent allocations. At this point it would be an error to reference the pointer again.

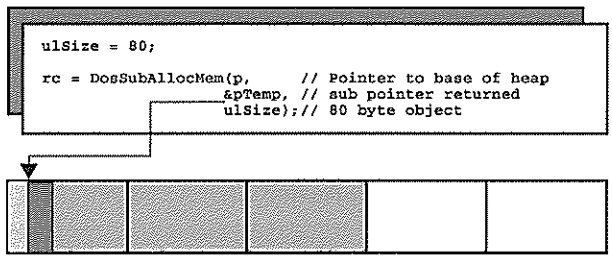
```
ulFlags = DOSSUB_INIT;
ulSize  = 4096;

rc = DosSubSetMem(p,       // Pointer to committed 12K
                  ulFlags, // Initialize a new pool
                  ulSize); // Initial pool size 4K
```



Allocating memory with DosAllocMem() is good for medium to large sized memory objects that would span multiple pages. Very large objects should be allocated as sparse, with pages committed as necessary to reduce overhead. However, DosAllocMem() is not good for small allocations that are less than one page in size. Since the size of each object allocated with DosAllocMem() is rounded up to the nearest multiple of pages, using this call for small objects can waste a great deal of memory.

In these cases, OS/2 provides a flexible memory suballocation capability. A larger memory object can be used as a heap, with smaller allocations taken from free space in the larger object. The memory must be prepared using the DosSubSetMem() API function. DosSubSetMem() accepts in its first parameter a pointer to the object which will be used for suballocation. The second parameter to DosSubSetMem() is an option flag which indicates how the object is to be prepared. A flag value of DOSSUB_INIT initializes the object, and reserves 64 bytes from the base of the object for OS/2 to control the suballocation work. This area would be used, for example, to maintain a list of free areas within the memory pool. Finally, the third parameter to DosSubSetMem() is the initial size of the pool, in bytes. This size should be smaller than the total size of the memory object. Then, the pool can later be increased in size by calling DosSubSetMem() again with the flag DOSSUB_GROW.

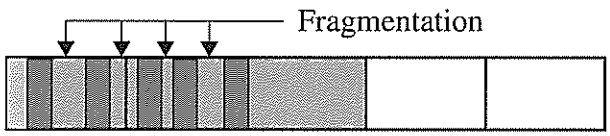
```
ulSize = 80;

rc = DosSubAllocMem(p,      // Pointer to base of heap
                    &pTemp, // sub pointer returned
                    ulSize);// 80 byte object
```



Once the memory pool has been initialized with DosSubSetMem(), DosSubAllocMem() may be used to carve out a small area of memory. A pointer to the base of the pool is passed in the first parameter to DosSubAllocMem(), and a pointer to the beginning of the suballocation is returned as a call by reference in the second parameter. The size of the suballocation is passed in the third parameter to DosSubAllocMem(). DosSubFreeMem() is used to release the suballocated memory. This API call also accepts three parameters: a pointer to the base of the pool, a pointer to the area being released, and the size of the area being released.

As calls to DosSubAllocMem() and DosSubFreeMem() are made to the same pool, it is possible for the memory in the pool to be fragmented since each sub-allocation must be contiguous in the pool. OS/2 provides no cleanup capability for this. DosSubAllocMem() simply allocates memory on a first-fit basis from the beginning of the pool, and this scheme works well for sub-allocation of similar size, as in the case for linked list or B-tree structures. The first-fit algorithm also allows the pool to be allocated as sparse memory, so that pages may be committed as necessary to handle sub-allocation when they occur. It is interesting to note that the C Set++ run-time library performs suballocation in order to satisfy memory requests made using the malloc() function.

```
ulSize = 80;

rc = DosSubFreeMem(p,       // Pointer to base of heap
                   pTemp,   // sub pointer to free
                   ulSize); // 80 byte object to free
```



In the sample program browse1, DosAllocMem() is used by the secondary thread to allocate enough memory to hold the file selected by the user. In this example, the memory is allocated as committed and writable, so that it is immediately available for the DosRead() API function to read the file. When the function main() has finished displaying the file, it calls DosFreeMem() to release the memory. The C library function malloc() is used to allocate storage for the small data structure used to share the pointer, file size, and semaphore handle between main() and the secondary thread. This data structure is released using the library function free().

OS/2 File Functions

OS/2 provides API functions to access files on disk, isolating the programmer from the details of the underlying file system. FAT or HPFS files may be accessed in identical fashion.

DosOpen() is used to create or open a file. The file name is specified in the first parameter. It can be just a file name in the current directory of the process, or a path and file name. This parameter is a string of up to 255 characters (the maximum file or path name length in FAT or HPFS). In the second parameter, OS/2 returns a handle to be used for all future references to the file. In the third parameter, OS/2 returns an integer indicating the action taken as the file was opened (for example, whether a new file was created or an existing file opened.) The fourth parameter specifies an initial size for a new file being created; this parameter is ignored if an existing file is being opened. File attribute flags may be specified in the fifth parameter, again, these only apply to a new file being created. The sixth parameter indicates a combination of actions requested. This could request either creation or failure if the file does not exist,

or, if it does, either open or failure. The open mode is specified in the seventh parameter. This could indicate the access permission (read/write, for example) requested by the process, as well as sharing permission (deny other processes write, but allow read). The last parameter to DosOpen() may be used to specify a buffer of extended attributes for the file.

```
HFILE hFile;
CHAR szName[] = "c:\\config.sys";
ULONG ulAction, rc;
...
/* Open an existing file to read */

rc = DosOpen(szName,      // File Name
             &hFile,      // File handle returned
             &ulAction    // Action performed
             0,           // No initial size
             0,           // No initial attributes
             OPEN_ACTION_FAIL_IF_NEW |
             OPEN_ACTION_OPEN_IF_EXISTS,
             OPEN_SHARE_DENYWRITE |
             OPEN_ACCESS_READONLY,
             0);          // No EA's
```

DosQueryFileInfo() may be used to determine more information about the file. This function accepts four parameters, beginning with the file handle to be queried. The second parameter is a flag indicating how much information is needed. FIL_STANDARD, for example, would request level 1 information which contains only the file's creation and last access time stamps, and the file size. Information levels 2 and 3 provide more information about the file's extended attributes. A pointer to a memory buffer, and the size of the memory buffer, are specified last. The buffer is filled with a data structure containing the information requested by level.

```
FILESTATUS fstsFile;
...

rc=DosQueryFileInfo(hFile, //Query file information
                    FIL_STANDARD;
                    &fstsFile,  // File status structure
                    sizeof (FILESTATUS));
```

All or part of a file may be read into memory using DosRead(). The file handle to be read is passed in the first parameter, and a pointer to a memory area is passed in the second parameter. The number of bytes requested to be read is specified third, and the number of bytes actually read is returned as a call by reference in the fourth parameter.

```
PVOID  p;
ULONG  ulBytes;
...

rc=DosAllocMem(&p, fstsFile.cbFile,
               PAG_WRITE | PAG_COMMIT);

rc=DosRead(hFile,    // FIle handle to read
   p,                // Pointer to memory buffer
   fstsFile.cbFile,  // # of bytes to read to buffer
   &ulBytes);        // # of bytes successfully read

rc=DosClose(hFile); // Done reading - close file
```

DosWrite() works the same way as DosRead(), writing the contents of a memory buffer pointed to by the second parameter, to the file handle specified in the first parameter. The number of bytes requested to write is in the third parame-

ter, and DosWrite() returns the number of bytes successfully written in the fourth parameter.

DosClose() closes a file. The only parameter to this function is the file handle to be closed. After this call is complete, the file may no longer be used unless it is re-opened.

In the sample program browse1, all file management is done by the secondary thread. The thread opens the file for read only, and will fail if the file does not already exist. DosQueryFileInfo() is used to determine the file size, from level 1 information requested. This size is passed to DosAllocMem() to allocate sufficient memory to hold the file. DosRead() is used to read the entire contents of the file into the memory buffer, so that it may be displayed as a whole later. Finally, the file is closed with DosClose(). If any of these calls should fail (a non-zero return code indicates failure) the thread notifies the function main() of this failure by indicating a zero file size in the shared data structure. If none of the calls fails, the pointer to the memory buffer and the size of the file are passed back to main() via the shared data structure, so that main() can display the file contents.

PM Coding

- Programming for the OS/2 Presentation Manager is very different than standard C programming. The complexity of OS/2 itself adds to the already steep learning curve. The following pages explain the basics and concepts of PM programming and should help new PM programmers get started.

- Introduction and Concepts

- PM Basics and Message Flow

- Function of main() in a PM Application

- PM Window Classes and Window Creation

- PM Window Procedure and Messaging

- PM Output and Window Painting

- Window Data Encapsulation

- Menu Resource Management

Introduction and Concepts

Programming for the OS/2 Presentation Manager can, initially, appear to be a very complex, involved procedure. The vast number of API functions alone is daunting, and the potential for error appears great. These perceived difficulties need not be the case. Presentation Manager is no more difficult than any other environment, only substantially different due to its use of an Object Oriented programming model. Although it is not a pure Object Oriented environment (according to most standard definitions) Presentation Manager does make use of many terms and concepts from Object Oriented theory. Keeping these concepts in mind, and adhering to a few basic principles, will make Presentation Manager programming a much easier and more productive task.

First, it is worthwhile to define several terms as they apply to the PM environment. Object Oriented terminology begins, naturally, with the object. A useful definition of an object is an entity which encapsulates both data and function. This contrasts with traditional, structured programming models in which a sharp distinction is drawn between code (function) and data (structures, variables, files, etc.) In theory, any function could act upon any data. This lies at the root of "side effect" errors in a typical program. In the object oriented model, data is kept absolutely private to a specific, well-defined set of functions that define an object's interface. These functions are often called the method of the object. The only access to an object's data is through invocation of a function defined in the method of the object.

For example, think of a car as an object. Every car encapsulates certain pieces of information - speed, mileage, direction, fuel level, engine temperature, etc. This data may only be modified through specific functions defined to the car object - start the engine, engage the transmission, accelerate, turn, etc. There is no way to alter a car's mileage without driving it.

To continue the car analogy, one does not need to be a mechanic to know how to drive. A function defined in an object is invoked through some very simple message or event. The message or event just notifies the object that the function must be invoked, and possibly passes some information to the object. A driver need know nothing of ignition systems, wiring, fuel injection, or spark plugs to know that a turn of the key will start the engine. Simple events like turning the key, turning the steering wheel, or pressing the pedals invoke much more complex functions that cause the car object to change state. One benefit of this architecture is that the messages form a consistent interface across multiple objects. All cars respond to essentially the same messages, so learning one car enables the driver to drive many other cars as well.

In PM, the window is the fundamental object. The window may encapsulate state information such as visibility, size, and position. The most important data for a window is likely its application-specific information that affects what it displays to the user or how it behaves in the context of a larger process. For

example, a text editor window would encapsulate perhaps the file name that it was working on, the foreground and background colors, and fonts, among other information. This data is accessed through a function known as the window procedure, which is invoked by PM itself in response to messages in the system, most of which result from user activity. For example, a simple message known as WM_PAINT invokes a window's window procedure, causing it to refresh its contents on the screen. This can involve very complex graphics routines working with encapsulated state information about the window. Most of this complexity is isolated from the user, and often even from the programmer as well. Of course, all windows need to be able to draw their contents to the screen, and this is invoked by the same message no matter what the window.

In true object oriented theory, every object belongs to a class. A class is defined as an abstraction, a description of common function (behavior) and possibly common information across a set of objects. This set may be empty; normally the programmer begins by defining a class and then creating individual objects as instances of the class when necessary.

Cars, for example, are often classified by size as sub-compact, compact, mid-size, and luxury. A specific car may be an instance of the class subcompact, which is defined by wheelbase, engine size, and other features. But sub-compacts and luxury cars still belong to the more general class of car, which may be defined as a four-wheeled self-propelled vehicle. Then sub-compact is said to be a subclass of the class car. This class hierarchy can be extended almost indefinitely; the class car might be a subclass of vehicle (boats, airplanes, etc.) which is a subclass of machine. Each level of the hierarchy is more abstract than the one below.

An important side effect combining messages and classes is polymorphism. This only means that objects of different classes may respond to the same message, but in different ways. A subcompact and a Formula 1 sports car both respond to the accelerate message, but performance is vastly different in each class.

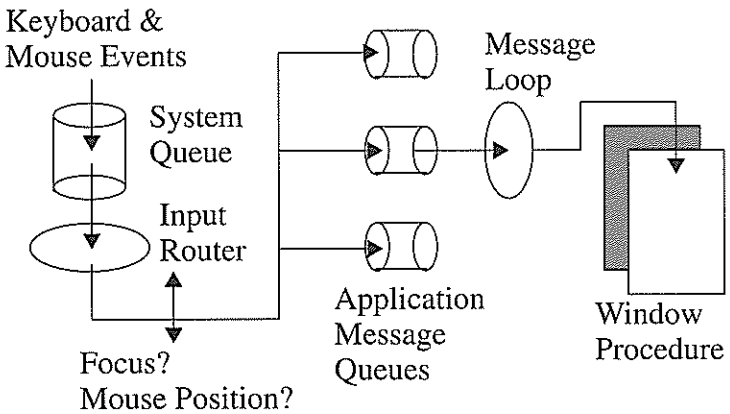
In PM, every window belongs to a class. There is no defined class hierarchy; just one level of abstract class definitions and then windows created as instances of particular classes. The window class defines the window procedure, so that in fact all the functionality for a window is drawn from its class. This, as we shall see, will have important repercussions regarding data encapsulation to individual windows. A window class in PM is defined by a class name which is associated to a specified window procedure, as well as some default style flags that determine rudimentary behavior of the windows. Most of the components of a standard PM window are in fact individual windows of PM-defined classes. For example, the title bar and the system menu are separate windows; both belong to PM-defined classes with their window procedures in DLLs provided with PM. Window classes enable polymorphism in PM as well. For example, a double-click action on mouse button 1 is a message that is received by whatever window the mouse is on at the time. If the user

double-clicks on the title bar, this has the effect of maximizing the entire aggregate window (or restoring it from maximized). If the user double-clicks on the system menu, this causes the aggregate window to be destroyed. The same message has different effects on different windows.

It will be important to keep these object oriented definitions in mind throughout the following discussion of the PM architecture and application model. A thorough knowledge of how messages are generated, how they cause invocation of the window procedure, and how they are dealt with from there, forms the key to understanding Presentation Manager programming.

PM Basics and Message Flow

To understand the structure of a PM application, it is important first to understand the architecture of the PM messaging system. Since this is a multitasking, multi-process environment, no application may be permitted to "own" the keyboard or mouse to retrieve input. (The display must also be shared for output, this will be discussed later.) Therefore PM operates in an event-driven manner. Interrupts generated by the hardware subsystems or device drivers are picked up by PM and translated into data structures representing messages. These are then queued for delivery to the applications, on a per-window basis. Other messages are generated directly by PM or by individual windows, and these are delivered directly to the application's queue. When a window receives a message, it invokes the associated window procedure which analyzes the message and takes whatever action is necessary in response.



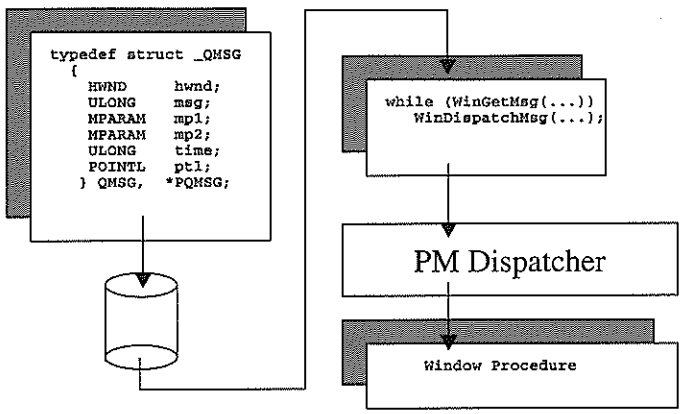
If an application generates a message, it always identifies the target window to receive the message. However, PM must determine the window receiving the mouse and keyboard messages. This is fairly straightforward, with a few minor exceptions. First, exactly one window on the desktop is always identified as having input focus. The user may identify the focus window by clicking on it with the mouse; the window may then show a cursor or display a selection rectangle to indicate that it has focus. If an application client area has focus, the title bar and frame are colored. Keyboard messages, in general, are destined for the focus window. Some keystrokes, however, have system-defined meanings.

. Ctrl+Esc, or Alt+Esc, for example, cause the operating system to change focus between applications. These keystrokes are trapped by PM and not delivered to application windows. Mouse messages are always delivered to the window the mouse pointer is on, with the exception that any application may temporarily capture mouse messages by calling the WinSetCapture() API function.

The input router delivers messages to applications one at a time; that is, it will not retrieve another message from the system queue until the last message has been dealt with. This is to ensure that focus changes between windows are handled correctly. This will have important consequences for multi-tasking in a PM application, since an application could lock out the entire system by not responding to its message queue.

The input router delivers the mouse and keyboard messages to an application-defined message queue. Other inter-application messages (DDE conversations, for example) or system-generated messages (to repaint the screen) are also placed in the application message queue. The application must create the message queue for itself; the existence of a message queue defines a PM application. The application will retrieve messages one at a time from this queue, and they will then be processed by the window procedure associated with the destination window. Without a message queue, most PM API functions are unavailable. In a PM application, the primary thread must have a message queue to deal with user input. Secondary threads, however, may also create their own message queues if necessary.

Throughout the process so far, every message is represented as a data structure. This structure is defined in the header file pmwin.h, as data type QMSG. The structure has six members, of which the first four are most important. The destination window handle of every message is the first member of the structure, and is of data type HWND. A handle uniquely identifies some object to the system, and is really defined as an unsigned long integer. Thus the window handle is nothing more than a 32-bit system-wide unique value identifying a particular window. The handle is assigned at the time the window is created. The second member of the QMSG structure is of data type ULONG, an unsigned long integer again. This member is normally a system-defined or application-defined constant that identifies the message. WM_CHAR, for example, defines keyboard messages and WM_BUTTON1DOWN is the message that occurs when button one of the mouse is pressed.



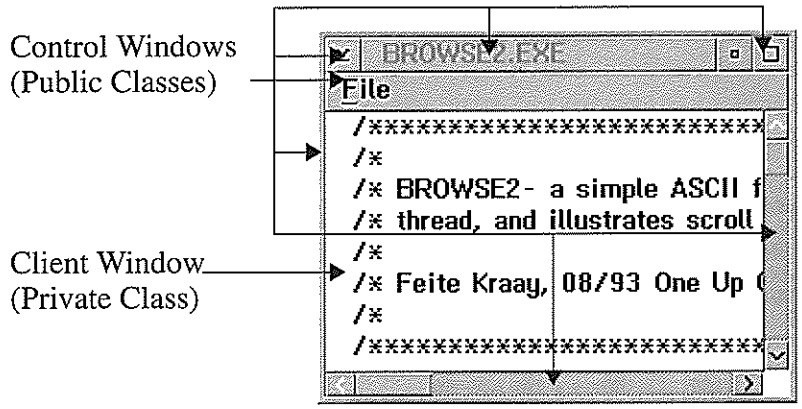
The next two members of the QMSG data structure are message parameters, of data type MPARAM. Although the actual data type for MPARAM is pointer to void, the contents of the message parameters vary depending on the message. Characters, long or short integers, window handles, pointers, all may be packed into or retrieved from the parameter. In a mouse message, for example, the first message parameter, mp1, will contain two short integers representing the mouse position. In a character message, the second parameter, mp2, contains the ASCII value of the keystroke that was hit. These parameters will be dealt with in more detail later. Finally, the QMSG data structure contains the time and mouse location when the message occurred.

The application must retrieve message structures from its queue, and deliver them for processing by window procedures. This is the job of the message loop, which consists of the two API functions WinGetMsg() and WinDispatchMsg(). WinGetMsg() retrieves a single message from the queue, or blocks if the queue is empty. When a message is received, WinDispatchMsg() passes it on to PM for processing by the window procedure. WinDispatchMsg() will not return until the message has been processed by the window procedure.

PM determines, from the window handle in the QMSG structure, what window procedure ought to be invoked. This is a fairly simple procedure. When a window is created, it is always created as an instance of a class. In true object oriented style, every object must belong to a class. When a class is defined or registered to PM, its associated window procedure is identified. Thus when a window is created, PM records in an internal window table the window procedure that will service the window. When a message is dispatched, the window procedure is determined by a look up in this window table. WinDispatchMsg() is an indirect invocation of the window procedure by the application.

Finally, the window procedure is invoked to deal with the message. This may be an application-defined window procedure, but in many cases will actually be a PM-defined window procedure. Every window the user deals with is actually assembled from many component windows -- the frame, title bar, menu, scroll bars, and others. Each of these windows, known as controls, is of a PM-

.defined window class that is registered at system initialization and has a window procedure built into the DLL file PMWP.DLL. Messages destined for these windows flow through the queues and message loop as described above, but are then passed to the control's window procedure in the DLL. The client area within the frame is usually application-defined, of a window class and window procedure private to the application. PM then invokes the application's client window procedure, calling back into the application's .EXE if necessary.



The window procedure must identify the message received, and take whatever action is necessary. A Title Bar window procedure, for example, will respond to the message identified as WM_BUTTON1DBLCLK by causing its associated frame to be maximized. The client area of a text editor will respond to WM_CHAR by displaying the character entered by the user. A much more detailed description of the window procedure will be the subject of a later section.

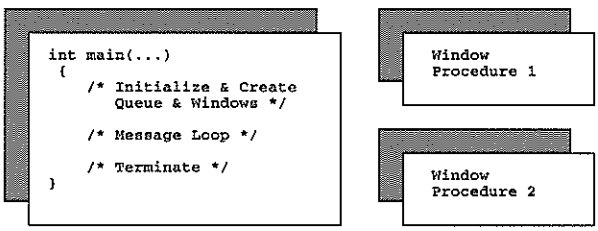
It is important to note that at several steps along this message flow, processing is synchronous. Beginning at the bottom, the WinDispatchMsg() API function is really a synchronous call to a window procedure, albeit indirectly through PM. This function will not return until the window procedure returns from identifying and dealing with the message. WinDispatchMsg() is in a loop with WinGetMsg(), so another message will not be retrieved from the queue until the current message has been completed. Also, PM will not retrieve another mouse or keyboard message from the system queue until the application returns to call WinGetMsg(), in other words, when the application has finished dealing with its current message. This means that if an application takes a long time dealing with a message in its window procedure, there is a real risk that the user is locked out from the entire system. To avoid this risk, a rule of thumb that PM applications should adhere to is that the window procedure must deal with each message it retrieves in under 1/10 of a second, although some documents are more lenient and allow 1/2 of a second.

Can a database search, file load, or spreadsheet recalculation be guaranteed to complete in so short a time? Of course they cannot, but the window procedure may certainly still return within 1/10 of a second as long as the programmer is careful to make judicious use of threads. For every message received in the

window procedure, the programmer must decide if the processing for the message may be too long. If it is, then a thread function should be invoked to perform the long job. As soon as the thread is started, control is returned back to the window procedure -- which can return it back to the message loop, which is now free to retrieve another message from the queue. In this way the application's interface is free to deal with further user interaction. Although there may or may not be much other application function available while the thread is busy, at least the user is also able to switch focus away from the application and work with another set of windows entirely. Meanwhile, when the thread finishes, it must notify the application. Although a semaphore could be used, it is much better style for the thread simply to place a message on the application's queue. When the client window procedure is invoked for this message, it will know that the thread has finished. This message would normally be a user-defined message set up in the application's header file. Multi-threading is, therefore, essential to a good PM application, to ensure its own responsiveness as well as that of the entire operating system, to user requests.

Function of main() in a PM Application

A PM application may be divided into two distinct components. The function main() does all the initialization and termination work, as well as the message retrieval from the application's queue as described above. The second component consists of one or more window procedures, which provide specific functionality for application-defined windows. The function provided by main() is almost the same across all applications. This section will examine in detail the standard API functions executed here.



First, main() must perform its initialization work. This includes variable declarations and header file include statements, as well as the creation of window objects and registry of window classes, if necessary. Normally, an application will need additional function above and beyond that provided by standard PM-defined window classes. This is accomplished by registering a new window class (that identifies a new window procedure) and then creating windows from the new class.

```
#define INCL_WIN          // Include PM header info.

#include <os2.h>          // Include OS/2 headers
#include <stdlib.h>       // C language headers
#include "browse2.h"      // App. private headers

int main (int argc, char *argv[])
{
  HAB hab;                // Anchor Block handle
  HMQ hmq;                // Message queue handle
  HWND hwndFrame;         // Frame window handle
  HWND hwndClient;        // Client window handle
  QMSG qmsg;              // Queue message structure
  ...
```

A PM application must include standard header file information from OS/2. As was discussed previously, the statements *#define INCL_WIN & #include <os2.h>* direct the pre-processor to include only the OS/2 header files necessary to provide prototypes and definitions for the PM window management API functions. The necessary #define statement for each API function is documented in the PM on-line reference. Other global header files may also be included, such as C library headers. Naturally, private header files to the application are included as well. Function prototypes for the window procedure and other functions, private data type definitions for encapsulated data, and constant definitions for user-defined messages would all be specified in private header files.

Next, main() must declare some standard variables. The data types HWND and QMSG have already been discussed. Two window handle variables are declared. A standard interface window created by a PM application is composed of a frame (whose visible attribute is the size border, and who creates in turn the other controls such as the title bar, menu, etc.) and a client window, which provides application-defined function. Each window is created individually, and identified with its own handle. The QMSG data structure is used to hold each message retrieved from the queue as it is dispatched for processing. HAB is a data type representing a handle to an anchor block. The anchor block can be thought of as defining the application's environment in PM, and will be discussed in more detail later. HMQ is the handle to the message queue. The application must create its own queue, and it is identified by a handle just as most other objects are identified by handles.

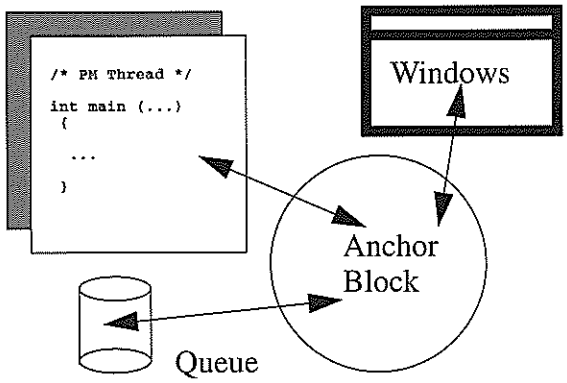
WinInitialize() is the first API function that a PM application must call. There is one parameter to this function; a flag that currently must simply be set to 0. This flag is documented to represent initial message-processing states for windows created in the application, but is not presently used in PM. WinInitialize() returns the anchor block handle, which must in turn be passed to other API functions that require it. The anchor block is used to identify the thread to PM for definition of private resources such as the message queue, and private window classes.

```
...
        // Create anchor block
hab = WinInitialize (0);
        // Create message queue of default size
hmq = WinCreateMsgQueue (hab, 0);

WinRegisterClass (...);

hwndFrame = WinCreateWindow (...);
hwndClient = WinCreateWindow (...);
...
```

Next, the application must call WinCreateMsgQueue(). This API creates a message queue for the main thread, returning the queue handle. The handle will be used in the termination processing, to destroy the queue. The anchor block handle is passed to this function, identifying the queue as being private to this thread. The second parameter to WinCreateMsgQueue() is the size of the queue, with a value of 0 indicating that a default size should be used. The default size is 10 messages, and this is sufficient for almost all threads.

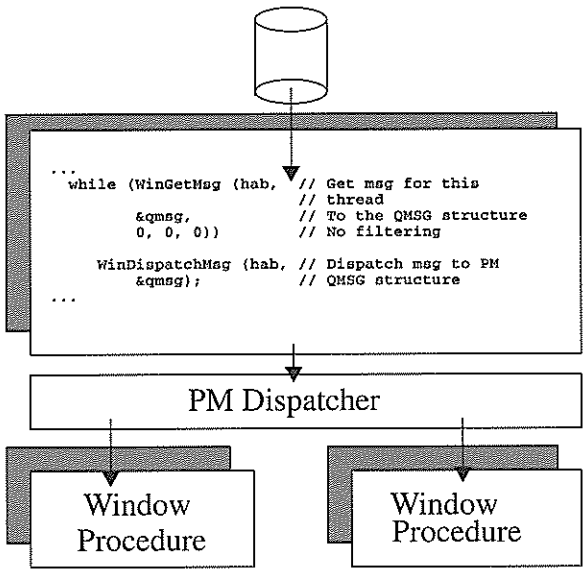
Then, the application must create its windows. The frame and the client are created individually, with calls to WinCreateWindow(). Optionally, they can be created in one step using WinCreateStdWindow(). In all likelihood, of course, the class for the client window must be registered first, using WinRegisterClass(). These calls will be discussed in more detail shortly.

The anchor block ties together the various objects created by a PM application. The primary thread, or main(), will create an anchor block. Other threads, if they need to have window management capability, must also create their own anchor blocks using WinInitialize(). Exactly one message queue must be created per anchor block, and it is private to the thread. Also, each window created by the thread is entered in the PM window table along with the anchor block handle (and message queue handle) where it was created. This way, PM is able to determine how to deliver messages. When a message is retrieved from the system queue, and PM has determined the destination window handle, the message must be placed on the application's queue. The queue handle can easily be found from the window table.



After the windows have been created and made visible, they will receive messages that originated with some user interaction. At this point the application must begin retrieving messages from its queue, using the function WinGetMsg(). WinGetMsg() will block if the queue is empty, and otherwise retrieve

the first available message into a structure of data type QMSG. The anchor block handle and the address of a QMSG structure are the first two parameters to this API function. The next three parameters define what is known as message filtering. If filtering is performed, then only messages that match a specific window handle and range of values are retrieved from the queue. Normally, in the main thread, filtering is not done so these three parameters may all be left as zero. WinGetMsg() returns the boolean TRUE whenever a message is retrieved, with one exception. The particular message identified as WM_QUIT indicates that the application must terminate, and if WinGetMsg() retrieves this message the function returns FALSE. Therefore, WinGetMsg() is embedded in a while() loop with WinDispatchMsg(); when the loop ends the termination work is performed.



WinDispatchMsg() accepts two parameters, the anchor block handle and the address of the QMSG structure retrieved by WinGetMsg(). WinDispatchMsg() passes the message on to PM for processing by a window procedure. When a window is created, it is identified as belonging to a class. The class in turn identifies the window procedure. A window table entry is made at the time the window is created, identifying its associated window procedure. This entry is retrieved by WinDispatchMsg(), and the window procedure is called. WinDispatchMsg() is synchronous; it will not return until the window procedure has completed processing the message. So, the message loop continually reads messages from the application's queue and deals with each one at a time through the window procedures. When WM_QUIT is retrieved, the loop ends and the application terminates.

All resources created or allocated by the application must be released. First, all windows that were created must be destroyed. This is done with the WinDestroyWindow() API function. WinDestroyWindow() takes one parameter, the window handle to be destroyed. As will be discussed shortly, all windows are created in a hierarchy defined by a parent-child relationship. In other words,

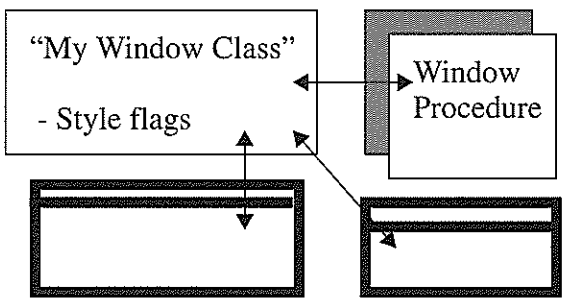
every window must have a parent. The frame window is the parent of the client as well as all controls such as title bar or menu. When a window is destroyed, all its descendant windows are destroyed as well. Thus, one call to WinDestroyWindow() specifying the frame window handle will often suffice to destroy all application windows.

```
   ...

   WinDestroyWindow(hwndFrame); // Destroy frame etc.
   WinDestroyMsgQueue(hmq);     // Destroy queue
   WinTerminate(hab);           // Destroy anchor block

   DosExit(EXIT_PROCESS, 0);    // End program
}
```

WinDestroyMsgQueue() accepts the message queue handle as its only parameter, and frees the memory for the queue. Then, WinTerminate() is passed the anchor block handle and releases all other resources allocated on behalf of the application by PM. At this point, there is little left for the application to do. Most PM API functions are unavailable without the existence of a message queue or anchor block. Usually, the application will just call DosExit() to terminate.

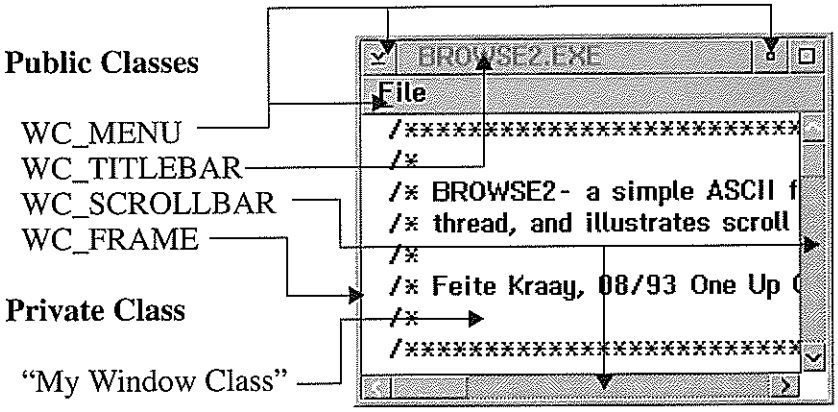
PM Window Classes and Window Creation

The window class, as has been mentioned before, identifies the behavior of a set of windows. In PM, the window class is defined by a set of style flags and a window procedure. The style flags identify basic characteristics of the windows, such as how re-sizing is handled or how re-painting interacts with child windows. The window procedure provides the message handling capability, and therefore the functionality, of the class. The class is named in PM with a character string. Every window belongs to exactly one class, and multiple windows might belong to the same class. This will have important implications for the window procedure, since it may have to service messages for multiple windows at the same time.



Window classes are of two types. Public Window classes are registered by PM at system initialization. These classes provide standard window capabilities to an application, such as title bars, menus, buttons, list boxes, etc. The class names for these public window classes are already defined in the header file pmwin.h. PM will already have created the window procedures for these

classes; the window procedures reside in the file PMWIN.DLL. For a standard application frame, windows of the public classes WC_MENU, WC_SCROLLBAR, WC_TITLEBAR, and WC_FRAME are normally created.



Public window classes provide standard function to a PM application. All title bars, menus, and scroll bars behave the same way, since they all share the same window procedure provided by PM. Although a public window's behavior may be modified by a process known as sub-classing the window, still, public window classes will normally not suffice for all activity in an application. The client area of a window must provide the application-specific functions and will therefore normally belong to a window class that is private to the application or thread.

A private window class has to be registered by the application when it initializes. This is done with the WinRegisterClass() API. WinRegisterClass() accepts five parameters. First is the anchor block handle, which indicates that the class is private to the thread where it was registered. Thus another application or another thread may register a class of the same name, without conflict. The classes will still be managed separately by PM. The class name is a character string defined within the application. It is used again in the WinCreateWindow() API to identify the class that is being instantiated there. Most important to WinRegisterClass() is the third parameter, the window procedure entry point. This associates the window procedure to the class, so that all messages for any window of the class will be dispatched to this window procedure. The style flags provide optional simple functionality for all windows of the class. CS_SIZEREDRAW, for example, ensures that a window will be fully redrawn every time it is resized. The last parameter to WinRegisterClass() specifies a number of bytes of additional storage to be allocated per window that is created from this class. This storage will be used to encapsulate application-specific instance data to the window, and will be discussed in more detail later.

BROWSE2.H

```
#define CLIENT_CLASS_NAME "WC_CLIENT"

MRESULT _System wpClientWndProc(...);
```

BROWSE2.C

```
...
WinRegisterClass(hab, // Register private class
                 CLIENT_CLASS_NAME, // Name constant
                 wpClientWndProc,   // Window procedure
                                    // entry
                 CS_SIZEREDRAW,     // Style flag
                 4);   // Four bytes of window words data
 ...

MRESULT _System wpClientWndProc(...)
{
   ...

}
```

One window procedure is identified when the window class is registered. Since multiple windows may belong to the same class, this function must be reentrant in order to handle all messages for any window of the class. This means that the window procedure should avoid the use of static variables, since any updates to a static variable would be reflected across all windows and would probably have harmful side effects in the application. The use of instance data by means of the storage requested with WinRegisterClass() is the preferred way to retain information for a window, and will be discussed in more detail later. Normally the window procedure is not invoked directly, but rather it is called indirectly through another PM API function such as WinDispatchMsg() or WinSendMsg(), both of which deliver a message to a window synchronously.

The window procedure is defined by PM for standard control windows, but must be defined by the application for private windows. This is normally the case for the client area. The window procedure is always defined to accept exactly four parameters, corresponding to the first four members of the QMSG data structure. When the application calls WinDispatchMsg() and passes PM a QMSG structure, PM then extracts the window handle, message ID, and message parameters as arguments to the window procedure that it calls. The window procedure is also defined to return a value of type MRESULT, which is itself just a pointer to void. MRESULT works the same way as MPARAM does; it is treated as a generic 32-bit value into which the application may place any information that should be returned to the caller. A normal default is to return 0 if the current message has been handled successfully and no additional information is required by the caller. The keyword _System identifies the window procedure to be using the system linkage convention in C Set/2. This convention, as for threads discussed earlier, is used for functions that will actually be called by the operating system rather than by application code.

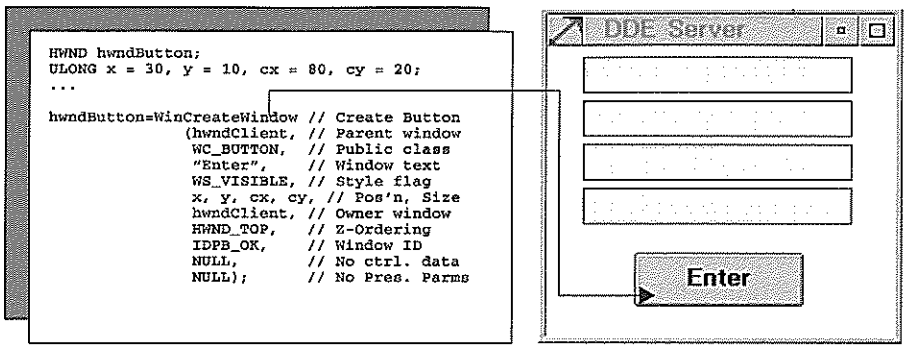
```
...
MRESULT _System wpClientWndProc(HWND hwnd, ULONG msg,
                                MPARAM mp1, MPARAM mp2)
{
  MRESULT mr = 0;

  switch(msg)                    // Test value of message
      {
        case WM_CREATE:
                                 // Do initialization work
             break;

        case WM_PAINT:
                                 // Do window drawing work
             break;
  ...

         default:                // Unwanted messages passed to
                                 // default window procedure
             mr = WinDefWindowProc (hwnd, msg, mp1, mp2);
             break;
       }
  return mr;
}
```

The body of the window procedure is simply a switch...case construction. The procedure tests the value of the message ID, and includes cases for those messages that are significant for application processing. WM_CREATE, for example, is a good message case for handling window initialization work since this message is passed to the window procedure at the time that a window is being created. WM_PAINT must be handled to ensure that the window's appearance on the screen is correctly maintained. All messages are documented in the *PM Reference* included in the Toolkit folders on the desktop.

Many messages passed to a window need not be explicitly handled by the application, but no message should be lost. PM provides an API function, WinDefWindowProc(), which is intended to handle all messages that the application does not. Thus a call to WinDefWindowProc() is the default case for the window procedure's switch statement. It is important to save the return value from WinDefWindowProc() since it may contain significant information that should be passed back to the originator of the message. The window procedure is normally built incrementally, writing and testing a few message cases at a time before continuing on to more messages.

A window, essentially, is an instance of a window class in an object-oriented sense. The window class provides basic styles and the window procedure, which are shared by all instances. The window object is the fundamental building block of a PM application. Windows are created for display purposes and also to provide "behind-the-scenes" function such as communication (DDE conversations, for example) or data storage (a window to manage database interaction). Not all windows will have visibility.

WinCreateWindow() is one of several API functions used to create a window. This call accepts 13 parameters, and is the most general window-creation function. A general example of this call is given first, to create an ordinary push button within a client window. Then, parameters will be discussed in detail to create a normal frame and client window combination.



First, WinCreateWindow() returns the window handle of the window being created. The value of the window handle is unique for each window in PM, and is used to retrieve class and queue information as has been discussed. In the 11th parameter of this call, the programmer may optionally also specify a window ID for the window being created. The window ID value is defined by the programmer, and must only be unique among all sibling windows. It is used by the application for internal message processing among its windows. For example, a push button will need to be distinguished from all other controls within the same client or dialog box. The ID would be used, for example, to determine which button generated a particular message. Thus, the window handle and window ID have very different uses in PM.

Whenever a window is created, the programmer must specify a window handle that represents the window's parent. This is the first parameter to the WinCreateWindow() API function. The parent-child relationship is used to define visible attributes of the window. For example, a child window is positioned relative to the bottom left corner of its parent, and is clipped within the boundaries of its parent. Thus a child window cannot be visible outside of its parent. Also, anything that affects the visibility of the parent (moving, destruction, etc.) also affects the child.

The window must be created as an instance of a specific class. The class name is specified as the second parameter to WinCreateWindow(). This may be a public window class (title bar, menu, or list box) or a private window class registered by the application. Constants are defined in the PM header files to represent the public window classes (WC_BUTTON, WC_LISTBOX, etc.) These constant names are documented in the PM on-line reference. An application may specify one of these constants and create a window of the corresponding public class, ready-made with the window procedure in a DLL provided by PM. To create the client, the application would specify the same class name used in the WinRegisterClass() API function.

Window text in the third parameter of WinCreateWindow() is optional, and not always used. A button window would display its window text, as would title bars and entry fields. Other windows will not display text. The text can be queried and modified with WinQueryWindowText() and WinSetWindowText().

Window style flags may optionally be specified in the fourth parameter of WinCreateWindow(), and are similar to the class style flags in WinRegisterClass(). Most of these affect basic painting behavior of the window. Most commonly, WS_VISIBLE may be used to make the window visible immediately upon creation.

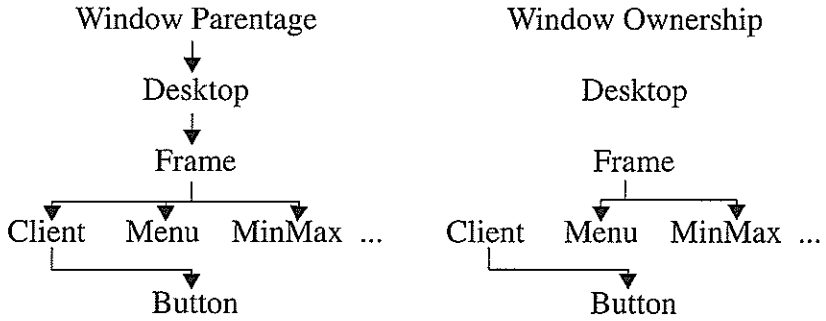
The window's position and size are specified, in pixels, relative to the bottom left corner of the parent. These are the next four parameters of WinCreateWindow(). The tenth parameter is Z-ordering, which identifies how the window overlaps with its siblings. Various constants are possible in this parameter, but HWND_TOP is normally used to indicate that the newly-created window should be displayed on top of any sibling windows that intersect with the same position. If HWND_TOP is not used, there is the danger that the window might be created but completely obscured by other windows, and therefore not actually visible.

Often, the size and position are left at zero initially, and the window positioned later using the WinSetWindowPos() API function. A button, for example, might be created by the client window during the WM_CREATE message case (when the client itself has no size or position yet) and positioned during the WM_SIZE message case (when the client itself is being resized.) Or, the frame and client are created initially with no size and position until other initialization work is complete; then they can be positioned relative to the size of the desktop.

A window may optionally have an owner. If it does, the owner window is specified in the ninth parameter of WinCreateWindow(). Ownership is only important in the context of control windows, and is used for notification messages. A list box, for example, will send a message to its owner window to indicate when the user has selected a line of text. A title bar will notify its owner (the frame) when the user is trying to move the window. Often an application's frame and client windows will have no owner.

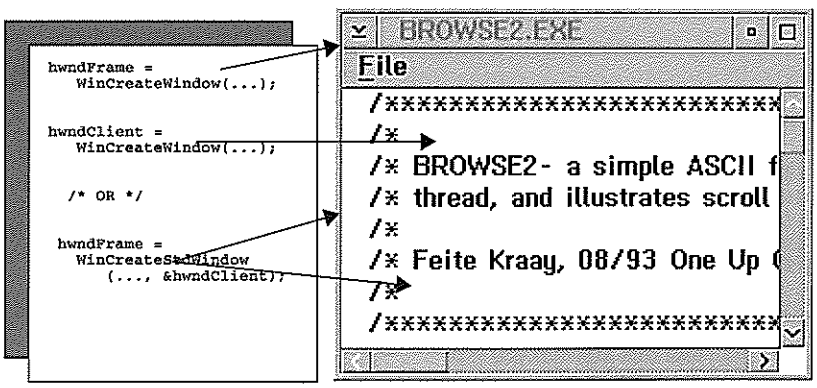
The last two parameters to WinCreateWindow() specify control data and presentation parameters, and these have different significance depending on the window class. These parameters are addresses of data structures. In the case of a frame window, for example, the Control Data structure identifies flags describing which additional controls should be created, and the handle to a DLL where other resource definitions such as the menu or icon may be found. For other controls, the Presentation Parameters structure defines the foreground and background colors, as well as the font to be used when displaying the window. For a general application client window, neither of these parameters is defined. They may then be used in an object-oriented way to provide application-specific initialization data to a window. This will be discussed in more detail in the *Window Data Encapsulation* section.

All windows are created within a strict parent-child hierarchy. HWND_DESKTOP is a constant defined in the PM header files to represent the window handle of the desktop, or screen background. An application frame window would be created as a child of the desktop, so that it could be fully visible on the screen. The frame will create several controls as children of itself, such as the title bar, system menu, minimize/maximize buttons, and the action bar. The client will also be created as children of the frame, and the programmer may wish in turn to create additional controls such as list boxes, buttons, or entry fields as children of the client in order to provide additional functionality. Remember, again, that every window must have a parent in order to be positioned and made visible.



The ownership hierarchy need not be complete. The frame will own other controls like the title bar, menu, system menu and min-max, while the client may own controls that it has created such as the list box and buttons. The frame controls send messages to the frame when significant user events occur, and the client's controls send it messages as well. Both the frame and client, however, may have no owner.

A PM standard window provides the basis for an application's interface. It is composed of a frame window and a client window. The frame window is the parent of the client, as well as a number of other controls such as the title bar, system menu, min-max, action bar, and scroll bars if necessary. The frame and client may each be created with WinCreateWindow() in separate steps, or WinCreateStdWindow() may be used to create both at once.



The frame window may be created by itself with WinCreateWindow(), as described above. The parent of the frame window is the desktop (represented by HWND_DESKTOP.) The class name for the frame is the constant WC_FRAME, defined in the PM header files. This represents a public window class that is standard for all frame windows. If the window text for the frame is not specified, then the title bar text defaults to be the name of the program. Otherwise, window text specified for the frame is displayed in the title bar.

```
#define ID_FRAME 100
...
HWND hwndFrame;

...

hwndFrame = WinCreateWindow     // Create Frame
                (HWND_DESKTOP, // Parent is desktop
                 WC_FRAME,     // Public class
                 NULL,         // No window text
                 WS_VISIBLE,   // Style flag
                 0, 0, 0, 0,   // No Pos'n, Size
                 0,            // No Owner window
                 HWND_TOP,     // Z-Ordering
                 ID_FRAME,     // Window ID
                 &fcData,      // Frame ctrl. data
                 NULL);        // No Pres. Parms
```

Often, the frame will be created with size and position set to 0. This is for ease of working with the client window. If the frame has a size, then the client, when it is created, must be positioned relative to the bottom left corner of the frame. The client window's position and size must be calculated to take into account the existence, and size, of the size border, scroll bars, menu, and title bar. Whenever the frame window is resized or repositioned, it automatically resizes and repositions all of its children including the client. Thus, it is much easier to create the frame and client with size and position set to 0. Then, one call to WinSetWindowPos() to position the frame window relative to the desktop will also position the client correctly.

The frame window need have no owner, and its window ID should be some arbitrary constant defined by the programmer. The second-last parameter, control data, is very important. This must be the address of a data structure initialized by the programmer to provide additional information to the frame window upon creation.

The control data supplied to the frame window is defined in a structure of type FRAMECDATA, which must be declared and initialized. The first member of the structure is just a count of bytes, the size of the structure itself. All PM-defined data structures begin this way. The second member of FRAMECDATA is a DLL module handle that identifies where resources may be located. When the action bar and icon are defined by the programmer in a separate resource file, this file may be compiled and embedded either into the PROGRAM .EXE file or into a separate .DLL file, to facilitate sharing among other programs. If the resources were built into a DLL, then the DLL must have been loaded with DosLoadModule() and the DLL handle passed to the frame window. Specifying 0 for the DLL handle indicates that the resources

were built into the .EXE file. Each of the frame's resources (menu, accelerator table, and icon) must be identified with the same ID number in the resource file, and that number specified in the third member of the FRAMECDATA structure. It is often convenient to use the same ID number as the frame window itself.

```
FRAMECDATA fcData;
fcData.cb = sizeof(FRAMECDATA);   // Size of structure
fcData.hmodResources = 0;         // Resources in exe, not in dll
fcData.idResources   = ID_FRAME;  // ID to find resourcesfc
fcData.flCreateFlags = FCF_SIZEBORDER | FCF_TITLEBAR   |
                       FCF_MENU       | FCF_MINMAX     |
                       FCF_SYSMENU    | FCF_VERTSCROLL |
                       FCF_HORZSCROLL | FCF_TASKLIST;
...

hwndFrame = WinCreateWindow ( ..., &fcData);
```

Finally, a set of Frame Creation Flags are combined with the bit-wise or operator and placed in the fourth member of the FRAMECDATA structure. These flags define the basic appearance and set of controls that the frame should have. These flags are documented in the on-line PM reference manual. One flag is set for each frame control window — title bar, system menu, etc. Another set of flags defines what kind of border the frame should have — size border, solid dialog border, no border, etc. Also, the flag FCF_TASKLIST should be specified to indicate that the frame should automatically add itself to the switch list maintained by PM. This means that when the user hits Ctrl+Esc, the application appears in the windows list. The address of the Frame Control Data structure is passed in the second-last parameter of WinCreateWindow().

Creating the client window requires another call to WinCreateWindow(). The frame window handle is specified as the parent of the client. The client window class is normally an application-defined class that was set up with WinRegisterClass(). If WinSetWindowPos() was used as described above, then the client's size and position are also all set to 0. A pointer to control data may optionally be passed to the client window in the second-last parameter.

```
#define CLIENT_CLASS_NAME "WC_CLIENT"
...
HWND hwndClient;
...
hwndClient = WinCreateWindow            // Create Client
                (hwndFrame,             // Parent is frame
                 CLIENT_CLASS_NAME,     // Private class
                 NULL,                  // No window text
                 WS_VISIBLE,            // Style flag
                 0, 0, 0, 0,            // No Pos'n, Size
                 0,                     // No Owner window
                 HWND_TOP,              // Z-Ordering
                 FID_CLIENT,            // Window ID
                 NULL,                  // No frame ctrl. data
                 NULL);                 // No Pres. Parms
```

The window ID for the client window is extremely important. This must be specified to be the constant FID_CLIENT defined in the PM header files. The frame window uses ID numbers to distinguish its various child windows, and these IDs are documented. FID_MENU represents the menu or action bar, FID_TITLEBAR the title bar, etc. When the frame needs to move or size its

children, or to pass messages on to its children, then the window ID is used to find the window handle of the appropriate child window. The frame always uses FID_CLIENT to represent the client window; if the client is created with a different ID number then the frame will not be able to determine its window handle and will not re-position or re-size the client, nor will it pass any significant messages (such as action bar activity) on to the client.

The window handle of any child can be determined from the parent window handle using WinWindowFromID(); this is what the frame does for each of its children. This is also useful for the client window to determine the window handles of its siblings in order to pass messages. The client may need to rearrange its scroll bars, or disable an action bar choice. These functions are accomplished by sending messages to the appropriate sibling window, and the window handle to which to send the message can be determined with WinWindowFromID().

To create a frame and client in one step, the application may call WinCreateStdWindow() instead of WinCreateWindow(). This function really just calls WinCreateWindow twice, to build the frame and the client, but it uses many default values. For example, size, position, ownership and Z-ordering cannot be specified with this function, nor may any initialization data be passed to the client.

BROWSE2.C

```
HWND hwndFrame, hwndClient;
ULONG flCreateFlags = FCF_ ... ;
...

hwndFrame = WinCreateStdWindow
                    (HWND_DESKTOP,        // Parent is desktop
                     WS_VISIBLE,          // Frame styles
                     &flCreateFlags,      // FCF_* values
                     CLIENT_CLASS_NAME,   // Private class
                     NULL,                // No title bar text
                     0,                   // Resources in exe
                     ID_FRAME,            // ID to find resources
                     &hwndClient);
```

The frame window handle is returned from WinCreateStdWindow(), and the address of the client window handle is specified in the last parameter so that PM can return this value as well. The parent window specified in this call should be HWND_DESKTOP, since it is the parent of the frame. WinCreateStdWindow() assumes that the frame will be the parent of the client. Then, frame window style flags (WS_* values) are specified if necessary (client window style flags are specified in the sixth parameter.) The third parameter is the address of a long integer that contains the frame creation flags as defined above. Next, the client window's class is provided (the frame window class is always WC_FRAME). Title bar text may be specified in the fifth parameter, but this text is appended to the name of the program first. The seventh and eighth parameters to WinCreateStdWindow() are the DLL module handle and ID number for resources, exactly as defined in the FRAMECDATA structure above. The ID number for resources is also used as the frame win-

dow's ID in WinCreateStdWindow(). This function provides no additional capability over WinCreateWindow() except that, having only nine parameters, it is a little easier to code.

In the example program, BROWSE2, the application's main() routine follows exactly the structure as outlined above. The anchor block handle and message queue are created with WinInitialize() and WinCreateMsgQueue(). Then, WinRegisterClass() is used to define a new window class private to the application. The class name is defined as a constant string in the header file browse2.h, and the window procedure is defined below in browse2.c. The class style flag CS_SIZEREDRAW is used to ensure that the window re-paints whenever it is re-sized. This is handy when, as in this case, the window has no icon or if the window's contents are dependent on size. Four bytes of data will be reserved per window created of this class, and these will be used to encapsulate information specific to the window, such as character and line sizes to manage the scroll bars.

The program uses WinCreateStdWindow() for the sake of simplicity, to create the frame and client window. The flag FCF_SHELLPOSITION asks PM to find a default size and position for the window. If this was not specified, the program would have to use WinSetWindowPos() to position the window explicitly. If a file name is passed as input to the program browse2, it will immediately start a thread to open and read the file. If not, it will wait for an action bar command to load a file.

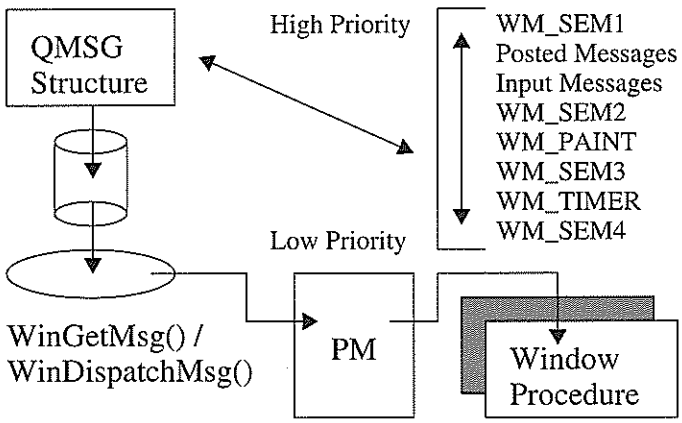
The standard message loop construction is used to retrieve messages from the application queue (WinGetMsg) and dispatch them to window procedures for processing (WinDispatchMsg).WM_QUIT, again, will cause WinGetMsg() to return the boolean FALSE, or 0, and the message loop will then terminate.

When the message loop terminates, WinDestroyWindow() is used to destroy the frame window. When a parent is destroyed, so are all of its children. The client and other frame controls are all destroyed too. Then WinDestroyMsgQueue() and WinTerminate() clean up any other resources still allocated to the application, before DosExit() is used to end the program.

PM Window Procedure and Messaging

Message handling in a window procedure represents the majority of work in a PM application. The main() function is similar across most applications. It is how messages are handled in a window procedure that distinguishes one application from another. The programmer must decide which messages to handle, and write the logic for each within a case statement in the window procedure. Hundreds of messages are already defined in PM, and it will often be necessary to define other application-specific messages beyond these. It is therefore important to understand which messages are significant in the window procedure, and what the purpose of each is.

Remember, again, that a message is represented in PM as a data structure of type QMSG. The first four members of this structure are passed as parameters to the window procedure, which tests the message ID value and determines what processing is necessary. This is a synchronous process; the next message is not read from the application's queue until the current message has been dealt with. Each message is identified by an ID number, which is defined either in the toolkit header file PMWIN.H or by the programmer in the application header file. Each message also includes two parameters of type MPARAM, which provide additional information relevant to the specific message. Before specific messages are examined in detail, it is worth noting the message flow through the application's message queue.



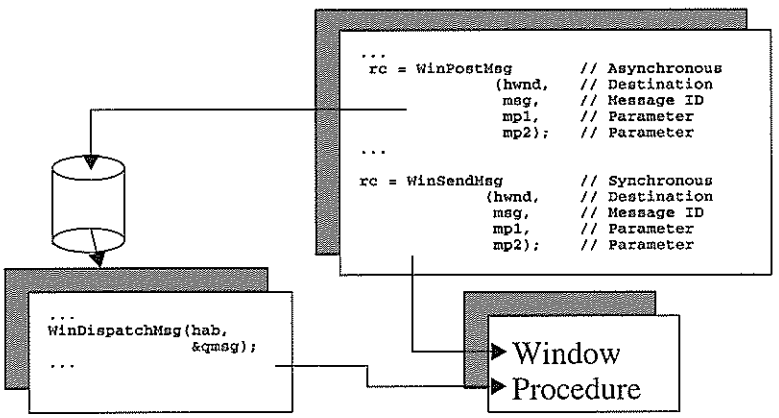
When messages are passed through the application's queue, they are not in a strict first-in first-out order. Rather, the messages are prioritized to improve performance. WM_SEM1 is defined to have the highest queue priority of all messages, and WM_SEM4 the lowest queue priority. They have no other meaning than that; but could therefore be used by the application to synchronize tasks. For example, an application may want to perform some task that requires use of the window's display area, perhaps to take a snapshot or screen capture. Anything else that affects the window's appearance should be allowed to complete first. Thus the snapshot could be taken when a WM_SEM4 message is dispatched to the window procedure. Since this is the lowest priority message, the programmer knows that when it is handled the queue must be empty. If the queue is empty, all other messages must have been dealt with.

Directly below WM_SEM1 in priority are messages posted to the application's queue by other threads or applications. (Some messages may be sent directly to the window procedure, this will be dealt with later.) These may be for a Dynamic Data Exchange conversation, or to indicate when a thread has finished processing. Command messages from the action bar, and other selected system messages, would also all appear at this priority. Below these messages are input messages, the mouse and keyboard messages that are retrieved by PM through the system queue.

WM_PAINT is a very low priority in the queue. This message indicates that a window's appearance must be refreshed (or painted in PM terminology.) PM considers window painting to be a high-overhead task, and so it is deferred for as long as possible. Placing WM_PAINT low in the queue ensures that other messages that may affect the window's appearance are handled first, and only the cumulative effect of all changes is drawn once under a WM_PAINT message. For example, if a window is partially in the background, yet exposed enough so that the user is able to double click on the system menu, it is expected that the window be destroyed. However, the mouse double click will cause the whole window to come to the foreground so that parts of it need to be repainted. Thus the WM_PAINT message as uell as WM_CLOSE (generated by the system menu in response to the double click) are placed on the queue. Since WM_CLOSE is of a higher priority, it is probably handled first (unless the queue is empty and WM_PAINT has already been retrieved.) If WM_CLOSE is handled first, then the window may be destroyed without the final, redundant paint work taking place.

WM_TIMER is also a very low priority message. An application may use the WinStartTimer() API function to start timer messages, which will be placed on the queue at the interval requested. However, since the messages are of low priority, they may not be handled by the window procedure until some time later. These messages are good for tasks which do not require a high degree of accuracy, such as flashing the cursor in an edit window. The highest accuracy timer services can be obtained from OS/2 using the DosStartTimer() API function.

WinDispatchMsg() is one way of invoking a window procedure and passing it a message to process. This function, as has already been discussed, is synchronous and waits for the window procedure to return from handling the message. The only place this function is used is within the message loop in an application's main() routine, or in a thread's message loop if it has one.

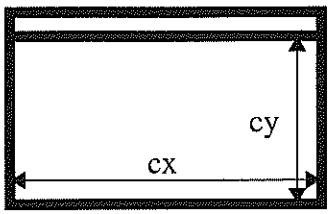


Outside of the message loop, a message may be passed to a window procedure either synchronously or asynchronously. In PM, if a message is synchronous it is said to be sent to a window; if a message is asynchronous it is posted to the window. WinSendMsg() is the API function that sends a message to a window.

This call invokes the window procedure directly, and waits for it to return. The four parameters to WinSendMsg() are exactly the four parameters to any window procedure — the window handle that received the message, the message ID, and the two message parameters. The return value of the window procedure is the return value of WinSendMsg().

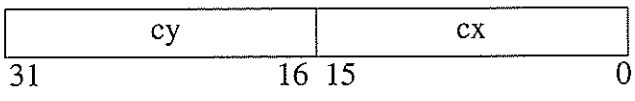
There are two reasons why a message may need to be sent instead of posted. First, the message may need to be handled immediately. For example, an application may want to disable a particular action bar choice while a thread is busy, perhaps because it would not make sense to have two copies of the same thread running simultaneously. Disabling a menu item requires sending a message to the action bar to change the item's attribute. This must be synchronous, to ensure that the user cannot start another copy of the thread. The message is sent, not posted. Also, in many cases an application may need to determine the state of some control. Whether a check box is on or off, or which line is selected in a list box, may influence other activity in the application. To determine the state of a control window, the application would send a message to the control. The control would set the return value (data type MRESULT) from its window procedure as an answer to the query — TRUE or FALSE for the state of a check box, or the number of a list box selected line, for example. Only when a message is sent to a window can the return value of the window procedure be retrieved, as the return value of the WinSendMsg() API function.

WinPostMsg() is used to pass a message asynchronously to a window. This is very similar to placing a letter in the mail. As long as the letter is in the mailbox, the sender may assume that it will be delivered to its destination, but it is not guaranteed to arrive in any specific period of time. When a message is posted to a window, it is simply placed on the queue for the thread servicing that window. Eventually, the thread will retrieve the message and dispatch it to the window procedure. Inter-application messages are usually posted (DDE conversations, for example). Also, in the above example for user-defined messages, the thread would post its message to the client window. Message posting is recommended whenever possible, since it avoids synchronous waiting and provides a more object-oriented feeling to the application; there are no time dependencies between the window objects. The four parameters to WinPostMsg() are the same as WinSendMsg(), the four arguments to the window procedure itself. WinPostMsg() returns a boolean flag indicating if the message was successfully posted.



WM_SIZE

mp1: old cx, cy
mp2: new cx, cy




```
USHORT cx, cy;
...
    case WM_SIZE: // Window has been resized
        cx = SHORT1FROMMP (mp2); // Extract new width
        cy = SHORT2FROMMP (mp2); // Extract new height
        ...
        break;
```

Each message is accompanied by two parameters which are of data type MPARAM. These 32-bit values are normally received as the formal arguments mp1 and mp2 in a window procedure, and may contain additional information relevant to the specific message WM_SIZE, for example, is sent to a window to notify it when its size has changed, and the new width and height are provided in mp2 as two short (16-bit) values. Thus the application must perform some type casting and bit-wise shift operations to extract these values from the message parameter.

For any message, each parameter will contain different data types, and therefore different operations will be required depending on the message. There are macros defined in the PM header files to assist with this; SHORT1FROMMP, for example, extracts a short integer from bits 0 to 15 of a message parameter. SHORT2FROMMP extracts a short integer from bits 16 to 31 of a message parameter. For each data type there is an appropriate macro. The parameter contents are documented along with the messages in the PM Reference manual. Other macros, such ar CHARnFROMMP (n ranges from 1 to 4), LONGFROMMP, PVOIDFROMMP, and HWNDFROMMP are also available to extract other data types from message parameters. Use of these macros isolates the program from relying on the actual underlying data types and thus makes the code more easily portable.

There are many standard messages defined in PM that must be handled by a PM application's window procedure. It is important to understand what these messages mean, the contents of the message parameters, and how they should be handled. All PM messages are documented in the on-line PM reference manual.

Message	mp1	mp2
WM_CREATE	Control Data	Pres. Parms.
WM_PAINT	0 - Reserved	0 - Reserved
WM_SIZE	Old cx, cy	New cx, cy
WM_BUTTON1DOWN	Mouse x, y	Hittest flags
WM_CHAR	Scan code & state	ASCII code
WM_TIMER	Timer ID	0 - Reserved
WM_COMMAND	Command ID	Source of cmd.
WM_CONTROL	Control Id, Code	Ctrl. Specific
WM_CLOSE	0 - Reserved	0 - Reserved
WM_QUIT	0 - Reserved	0 - Reserved
WM_DESTROY	0 - Reserved	0 - Reserved
WM_SEM1	Semaphore value	0 - Reserved
WM_USER	App. Defined	App. Defined

WM_CREATE is sent to the window procedure whenever WinCreateWindow() creates a window of the corresponding class. This message is sent before the window has been made visible. The two message parameters contain pointers to control data and presentation parameters; these are the same pointers that were the last two parameters to the WinCreateWindow() call itself. In the case of a client window, either of these parameters may contain a pointer to application-specific initialization data for the window — for a text editor, perhaps, the name of the initial file to display.

WM_CREATE is normally used to perform initialization work for a new window, since it is sent once only to any window upon creation. A text window may wish to initialize font and color information here. A graphics window might do some initial drawing work upon window creation, and apply transformations later to display the work when the window needs to be repainted. How this data, and data received via the pointers, is stored is the subject of a later section.

WM_PAINT is either sent or posted to a window, by PM, whenever the window's appearance needs updating. If the window's class was registered with the class style flag CS_SYNCPAINT (synchronous painting) then this message is sent, otherwise it is posted. The application must handle this message for its client window. It must also notify PM that the window's appearance is again valid. Painting, and window output in general, will be discussed in the next section, "PM Output and Window Painting"

WM_SIZE is sent to a window every time its size is modified by the WinSetWindowPos() API function. When the frame is resized by the user, it resizes the client window and WM_SIZE is generated. When this message is retrieved, the new width and height may be extracted from the second message parameter, mp2, using SHORT1FROMMP and SHORT2FROMMP respec-

tively. The client window may need to use this message case to adjust the position of other child control windows. The new size must also be used to adjust the size of the thumb mark in the scroll bars, if they exist.

Button and keyboard messages may be retrieved by a client window to handle direct user activity. Obviously, a text editor would need to handle WM_CHAR. A graphics editor would perhaps want to handle mouse messages to accommodate freehand drawing. The ASCII character value struck is provided in mp2 of WM_CHAR, and the pointer location relative to the bottom left of the window is provided in mp1 of WM_BUTTON1DOWN. These messages are posted to the application queue by the system input router. Other mouse messages of interest are WM_BUTTON1UP, WM_BUTTON1CLICK (down followed by up), and WM_BUTTON1DBLCLK. There are also, of course, the same messages for mouse button two as WM_BUTTON2*.

The WM_TIMER message is posted to the application's queue by PM, if any window has started a timer service using the WinStartTimer() API function. Timer messages, in fact, go through the system queue first just as the other hardware-based messages (mouse and keyboard) do. The first message parameter contains the timer's ID number, which is assigned using WinStartTimer() and must be below the constant TID_USERMAX defined in the PM header files. This way a window may start, and distinguish, more than one timer. Remember that timer messages are low priority in the queue, and so may not be handled by the window until some time after they are delivered to the queue.

WM_COMMAND and WM_CONTROL appear to be quite similar, but there are significant differences in how they are managed. WM_COMMAND is posted to the owner of an action bar (menu) window, or to the owner of a push button window. WM_COMMAND may also be generated by the accelerator table resource. The owner of the action bar is the frame window, and the frame sends the WM_COMMAND message on to the client window. The first message parameter of WM_COMMAND contains the ID of the action bar choice, or the ID of the button window, so that the recipient may distinguish the source of the message. Every menu or action bar option must have a distinct ID number, as will be discussed shortly.

WM_CONTROL, by contrast, is always sent from any control window (radio button, list box, etc.) to its owner window in order to notify the owner of significant user activity. The check box, for example, would notify via WM_CONTROL whenever it is clicked with the mouse, and the list box would notify if it is scrolled, or if a selection is made. The first message parameter of WM_CONTROL contains two short integers. The first (SHORT1FROMMP) is the window ID number of the control — all control windows must have distinct ID numbers, with respect to their parent window. The second integer (SHORT2FROMMP) is a notification code identifying specifically what happened to the control. BN_CLICKED, for example, would mean that a button was clicked with the mouse, or LN_SELECT would indicate that a list box line

was selected. Other information may appear in the second parameter of WM_CONTROL, but this would depend on the individual control window. In the on-line PM Reference manual, all control windows are documented along with each of their notification codes under WM_CONTROL.

WM_CLOSE, WM_DESTROY, and WM_QUIT are often confused, and it is important to distinguish their uses. First, WM_CLOSE is posted to a client window when the user selects close from the associated system menu window. This therefore signals the user's intention to close a particular window only. If the window in question is the application's main window, the application should normally ensure that any user changes have been saved. An editor, for example, should prompt the user if the latest modifications to a file have not been saved. Other applications simply prompt for confirmation with a message box, "Are you sure you want to end the program?"

If the client window procedure does not handle the WM_CLOSE message, WinDefWindowProc() will automatically post a WM_QUIT message to the same queue. This can be a problem if the user wishes to close a child window in the application. Since WM_QUIT will terminate the message loop, then closing a child window could close the whole thread or process. In this case, the child window procedure must handle WM_CLOSE itself, and call WinDestroyWindow() to destroy just the child's frame (and therefore all its descendants).

WM_QUIT is posted to the application whenever PM determines that the application should terminate. This could be due to the default message processing of WM_CLOSE as discussed above, or due to the user shutting down OS/2, or due to the user closing the application from the window list. A window would also post WM_QUIT itself after having confirmed via a prompt in the WM_CLOSE message processing. WM_QUIT causes WinGetMsg() to return the boolean FALSE, thus terminating the message loop. Then, the application's main windows and queue are destroyed, and the application is ended.

WM_DESTROY is sent to the window procedure whenever a window is destroyed with WinDestroyWindow(). This message, then, performs the opposite duty of WM_CREATE. An application should use the WM_DESTROY message case to do any clean-up work necessary on resources that were allocated for the window under WM_CREATE. Memory to hold instance data, for example, should be released here.

A rough sequence of messages for a typical window would begin with WM_CREATE, sent upon window creation so that the window procedure may do any initialization work necessary. Then, the window would handle any number of WM_CONTROL, WM_COMMAND, user-defined, mouse, and keyboard messages. Among these would also be WM_PAINT, to refresh the contents of the window on the screen, and WM_SIZE, to manage scroll bar and sizing of child windows. Finally, the user double-clicks on the system

menu, causing WM_CLOSE. The application in turn posts WM_QUIT, and when the window is destroyed WM_DESTROY allows the application to do final clean-up work for the window.

The PM-defined messages are not sufficient for all application communication. If a window procedure has, correctly, used a thread to perform a database search or file load, there is no message defined in PM that is designed specifically for the thread to notify the window procedure when it has finished. (In the meantime, the window procedure could be handling other messages. When it receives a message from the thread, it would then display the report to the user.) In situations like this, the programmer may define new messages, only taking care that the new messages do not conflict in value with any messages already defined in PM.

BROWSE2.H

```
...
#define UM_FILEREAD WM_USER + 1    // Success in thread
#define UM_FILEERROR WM_USER + 2   // Error in thread

VOID _System fnOpenThread(POPENINFO pOpenInfo);
...
```

BROWSE2.C - thread

```
...
WinPostMsg(hwndClient,             // Notify client of completion
           UM_FILEREAD,            // Successful read
           MPFROMP (pBuff), 0);    // File buffer in mp1

 DosExit(EXIT_THREAD, 0);
```

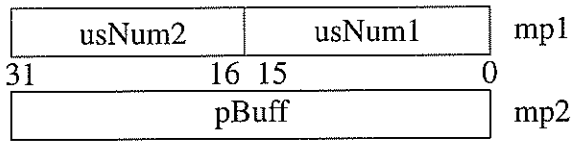
BROWSE2.C - window procedure

```
...
   case UM_FILEREAD:               // Display new file
        break;
```

WM_USER is defined to have the highest possible value of all PM messages. Any message number higher than WM_USER, but shorter than the maximum size of a long integer, can be defined in the application. Thus messages providing data to a client window, or notifying the window of thread termination, may be defined as WM_USER + 1, WM_USER + 2, etc. One convention is to use the prefix UM_ for these user messages, to distinguish them from regular PM window messages that begin with WM_. Of course, the parameters on the user messages are also application-defined, and should be documented in the header file where the messages are defined.

When an application generates a message of its own for another window, it may have to construct message parameters to go along with the message. Just as there are macros to assist in the bit-wise operations necessary to extract from a message parameter, so there are macros like MPFROM2SHORT to make the message parameter, in this case from two integers. The first argument to MPFROM2SHORT becomes the low 16 bits (0 - 15) of the parameter, and the second argument becomes the high 16 bits (16 - 31) of the parameter. MPFROMP, MPFROMLONG, and MPFROMHWND are other parameter building macros provided in the toolkit header files.

```
#define UM_DATA WM_USER + 5
...
PVOID pBuff;
USHORT usNum1, usNum2;
...
WinPostMsg(hwndClient,
           UM_DATA,              // Post user message
           MPFROM2SHORT (usNum1, usNum2),
           MPFROMP (pBuff));
```



In the sample program browse2, the client window procedure uses WM_CREATE to do initialization work for the scroll bars. The window must know what units to scroll by horizontally and vertically (character width and line height) and these are calculated from the current font. The window handles of the scroll bars are also retained.

In the WM_SIZE message case, a single function call to the application function fnFixScrollSize() ensures that the scroll bar thumb-marks are correctly sized. Management of the scroll bars will be discussed in detail later, along with the window painting done in the WM_PAINT message case.

The user's action bar selections are presented to the application under the WM_COMMAND message. Here the application determines which choice was selected, and takes the appropriate action: opening a file, clearing the window, or ending the program.

Two user messages are handled to receive communication from the thread. One message indicates that an error occurred handling the file requested, and the other indicates that the file has been loaded and is ready to be displayed.

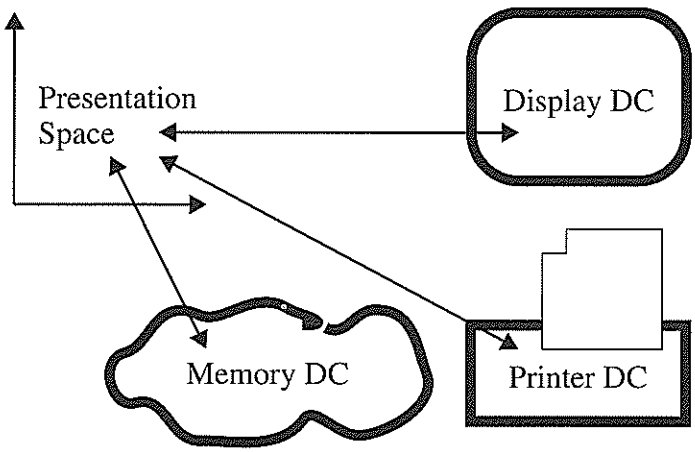
In the WM_CLOSE message case, the application simply prompts for confirmation from the user via the WinMessageBox() function. This API displays a message in a small dialog window, with a selection of push buttons and icons. If the user selects the yes button, then the application posts WM_QUIT to terminate. When the client window is destroyed, the WM_DESTROY message is then received by the window procedure. Here, the instance data memory is released, and if a file was displayed, the memory buffer containing the file is also released.

PM Output and Window Painting

By means of the message queue model, PM automatically shares all input devices — mouse, keyboard, and timer — among all applications. However, the screen must also be shared for output in PM. This implies some strict rules that must be followed for a PM application to draw to its window - drawing directly to the video buffer is, of course, disallowed. Although an application

may draw to a window at any time, as much drawing work as possible should be restricted to the WM_PAINT message case in the window procedure. Otherwise, care must be taken to avoid inconsistencies in the window's appearance if drawing work is done outside of WM_PAINT. An exception to this rule would be an application such as a graphics editor, which would draw shapes on the window as it tracks the mouse. These must then be retained and redrawn in WM_PAINT.

Four steps must be followed in order for any output to be possible in a PM environment. First, the application must open up a device context for the particular physical device to be used. The device context can be thought of as representing the device driver to the application. Device contexts can be opened for memory, metafiles, printers and other queued devices, and screen windows. The device context contains device-specific information — page size, font and color capabilities, and resolution, for example.



Second, the application must create a presentation space. The presentation space can be thought of as a scratch pad; a device-independent area to hold the application's drawing work. As far as the application is concerned, the presentation space is simply a set of cartesian coordinate axes with a particular resolution. The application may draw in any quadrant of these axes, but normally the origin (0,0) is mapped to the bottom left corner of the window or device. Thus in most cases, drawing is restricted to the upper left (positive x, positive y) quadrant, or transformed to fit in that quadrant.

The device context and presentation space must be associated. This can be done with an API call such as GpiAssociate(), or done with the GpiCreatePS() function directly as the presentation space is created. In fact, some API calls (WinGetPS(), for example) return a presentation space that is already automatically associated to a device context for the window, so these first three steps are often combined. Finally, the fourth step is for the application to draw to the presentation space. The drawing is then represented on the device. This allows a large degree of device independence in the application, since the drawing to the presentation space is done the same way no matter what device context is associated.

Three types of presentation spaces are generally available in PM. The fourth, AVIO (Advanced Video I/O), is only available as16-bit code. This presentation space provides only character mode output, and is not recommended for new PM applications. Three API functions are commonly used to create the graphical presentation spaces, and each has specific uses.

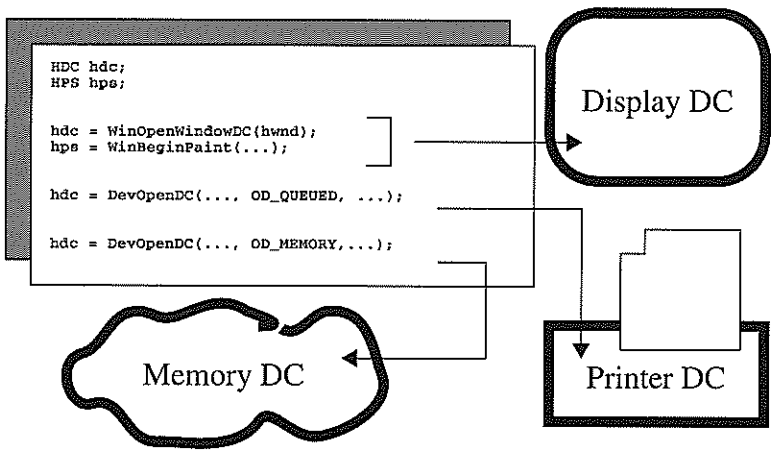
The cache-micro presentation space is normally allocated by an application using WinBeginPaint() or WinGetPS(). These presentation spaces are kept in memory by PM, and are already associated to window device contexts. Thus they are very fast to set up and use, but are restricted to screen output only. General window painting work would use the cache-micro presentation space. This presentation space must be released using WinEndPaint() or WinReleasePS(), since it is a limited PM resource. If an application does not release its cache-micro presentation space, then eventually all other windows in the system will be unable to repaint themselves.

PS Type	API to Build	Notes
Cache Micro	WinBeginPaint() WinGetPS()	Kept in memory by PM, low overhead, no retained graphics
Micro	GpiCreatePS()	Low overhead, no retained graphics
Normal	GpiCreatePS()	Higher overhead, full graphics capability
AVIO	VioCreatePS()	16-bit character mode only, no graphics

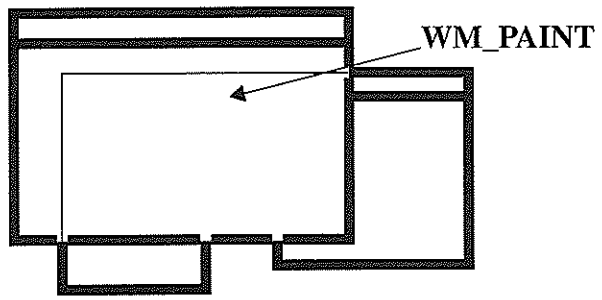
The micro presentation space is similar to the cache-micro in that both provide only restricted graphics functions. In particular, no retained graphics are possible with this presentation space. Retained graphics allow an application to hold graphical objects such as lines, arcs, and text in segments, which can later be transformed and drawn as a single unit. This can cause a significant overhead in system memory. In general, the micro presentation space is very fast to use. The micro presentation space may be associated to any device context. A graphics application would commonly use a micro presentation space with a device context for memory, to hold and process a bitmap before displaying it. The micro presentation space is created using GpiCreatePS().

The normal presentation space provides the greatest functionality along with the greatest overhead. This presentation space is also created with GpiCreatePS(), and may be associated to any device context. Full graphics functions are available, including retained graphics and transformations.

A device context for the screen window is created easily using WinOpenWindowDC(). WinBeginPaint() and WinGetPS() return micro presentation spaces already associated to window device contexts, so in these cases WinOpenWindowDC() is not necessary. For all other device types, an application would use DevOpenDC() to open up a device context, passing it a flag to indicate what kind of device context is needed. The flag OD_MEMORY would open up a device context of memory, allowing the application to prepare a bitmap (combine colors and images) before displaying it to the user. OD_QUEUED would represent any queued device set up in PM, normally a printer or plotter. Other flags are also available for the DevOpenDC() API call. OD_DIRECT allows the application to write directly to a printer or plotter and bypass the PM spool queue. This is not usually recommended since it may interfere with queued printing. OD_INFO is used to query a printer or plotter's page size, resolution, and other capabilities. OD_METAFILE creates a device context for a metafile, which contains the graphics drawings according to specific DCA (Document Content Architecture) standards.



In most cases, all the window's drawing work can be restricted to within the WM_PAINT message case. PM will generate and post a WM_PAINT message to the window whenever the window's appearance is invalid. This could happen if the window is in the background, and is being brought to the foreground or otherwise uncovered by other overlapping windows, or if the window is being resized. WM_PAINT may be sent instead of posted, if the window class was registered with the class style flag CS_SYNCPAINT, for synchronous painting. Preferably, however, WM_PAINT should be handled asynchronously. Then painting will only happen if absolutely necessary, since WM_PAINT is a low priority message in the queue.




```
HPS hps;
RECTL rclInvalid;
...
  case WM_PAINT:                          // Window invalid -
                                          //    needs repainting
    hps = WinBeginPaint(hwnd,             // Get PS for window
                        0,                // No PS already
                        &rclInvalid);     // Invalid rectangle

      // Do painting work

    WinEndPaint(hps);                     // Return PS to system
    break;
```

What if an application needed to refresh the appearance of a client window? Possibly, as a result of some user-initiated action the state of the window has changed, and it needs to be redrawn. The user may have selected a new file from the action bar, and this file must be displayed. Would the application just call WinPostMsg(), posting a WM_PAINT message to its own client window? The answer is no, since PM expects to control exactly how window painting is initiated. If an application simply posts or sends WM_PAINT, the message will not be handled correctly since it did not originate from PM in the first place. The reason is that PM only expects a window to be repainted if its condition is in fact invalid. PM maintains, for each window, a flag representing its visible state. If the window is resized, or uncovered, then PM invalidates the window and automatically generates a WM_PAINT message. If the application needs to force a repaint on any window, it must cause the window to be invalidated with the WinInvalidateRect() API function. This function accepts two parameters, a window handle and the address of a rectangle structure (data type RECTL) that describes the invalid area of the window. If the rectangle address is NULL, the entire window is invalidated.

WinBeginPaint() returns a handle to a cache-micro presentation space. The window handle to be repainted is passed in the first parameter, and the third parameter of this API is an address of a rectangle data structure. When the call returns, this call by reference has set up a rectangle that delimits exactly what portion of the window is invalid. This could bound a region that was previously covered by other windows, or it could be the rectangle that was specified in WinInvalidateRect(). The second parameter of WinBeginPaint() is a presentation space handle. Some applications may create and customize a presentation space upon window creation; then WinBeginPaint() will set it up to repaint the invalid area of the window and not return a new presentation space.

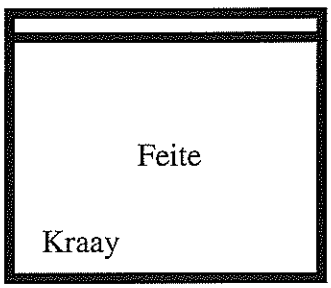
It is important to note that WinBeginPaint() can distinguish what is known as the invalid region of the window — that is, the collection of smaller rectangles within the window boundary that really need updating. The rectangle returned in the third parameter is also known as the bounding rectangle, since it is the smallest possible rectangle that encloses the entire invalid region. When the application draws using this presentation space, the drawing is only really performed if it intersects with the invalid region, the rest of the window being considered still valid from the last time it was drawn.

When the painting is complete, the cache-micro presentation space must be returned to PM. This is done with WinEndPaint(). If the presentation space is not returned, then eventually PM will run out of presentation spaces and other windows will be unable to paint themselves.

WinBeginPaint() is essential to correct painting in a PM application even if the cache-micro presentation space is not used. When the application paints, it could use a micro or normal presentation space, created with GpiCreatePS() instead. However, the application must still call WinBeginPaint(). The reason for this is that when WM_PAINT is received in the window procedure, the window is still considered invalid by PM. If after the window procedure finishes handling WM_PAINT the window is still invalid, PM will simply generate another WM_PAINT message and the application is stuck in a painting loop. WinBeginPaint() is the only API function that will validate the window to PM. Since WinBeginPaint() must be called, and a presentation space handle is therefore returned, WinEndPaint() must also be called. This, in fact, is the default window procedure handling of WM_PAINT. It just calls WinBeginPaint() to validate the window, and WinEndPaint() to return the presentation space, and does no drawing work in between.

All API functions that draw to a window require as their first parameter a presentation space handle. The simplest function is just WinFillRect(), which allows the application to fill a rectangle with a specified color value. WinFillRect() requires three parameters: the presentation space, the color, and the address of a rectangle data structure which is to be filled. Each color is represented in PM as a long integer, and constants are defined in the header files to represent the 16 colors present in the default color table, as well as some system-defined colors. Although it is possible for an application to color its window background with a specific value such as CLR_BLUE, it is better for the application to allow the user to configure colors. If the application paints with the system-defined color constant SYSCLR_WINDOW, this will represent the window background color according to the user's configurations from the system setup folder. If the application always paints with a specific color value, then that is much less easy to change according to user preferences.

```
HPS hps;
RECTL rclWindow;
POINTL ptlText = (10, 10);
CHAR szText1[] = "Feite";
CHAR szText2[] = "Kraay";
...

hps = WinBeginPaint(...); // Get PS for painting
                          // Query window dimension
WinQueryWindowRect(hwnd, &rclWindow);
                          // Fill background color
                          //     in rectangle
WinFillRect(hps, &rclWindow,
            SYSCLR_WINDOW);
                          // Draw text centred in
                          //     rectangle
WinDrawText(hps, -1, szText1, &rclWindow,
            SYSCLR_WINDOW,
            SYSCLR_WINDOWTEXT,
            DT_CENTER | DT_VCENTER |
            DT_ERASERECT);
                          // Move graphics cursor
GpiMove(hps, &ptlText);
                          // Draw single line of
                          //     characters
GpiCharString(hps, strlen(szText2), szText2);
```



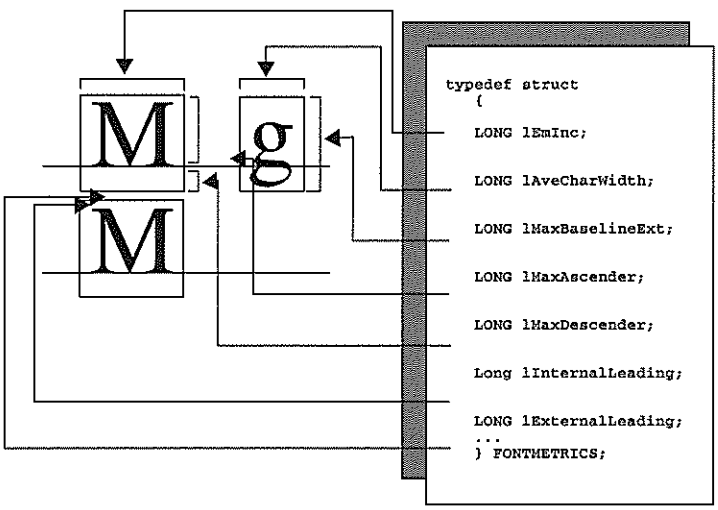
GpiCharString() may be used to draw a single character string anywhere in a presentation space. This array need not be zero-terminated, since its length is passed in the second parameter of the function. GpiCharString() draws at what is called the current location of the presentation space, which is the location of the graphics cursor. Initially, the current location is the origin of the axes that define the presentation space, but it is then moved to be the end point of the last object drawn. The function GpiMove() accepts two parameters, the presentation space handle and the address of a point data structure (POINTL, with members x and y) to update the current location to the point specified. GpiMove() should be called before GpiCharString() to ensure that the string is drawn at the right location. In fact, this is so common that the function GpiCharStringAt() is provided as well, with the point specified in the second parameter, followed by the length and the string. GpiCharStringAt() moves to the point requested, whether it needs to or not, before drawing the string.

WinDrawText() can be useful for drawing single lines of text. The text, again, need not be zero-terminated since its length may be passed in the second parameter. If the length parameter is set to -1, this indicates that the text is zero-terminated. WinDrawText() allows the programmer to select the foreground and background color to be used in drawing the text within a rectangle. GpiCharString() draws only in the foreground color as it was last set with GpiSetColor(). The last parameter to WinDrawText() is a set of flags defining how the text is to be drawn — centered in the rectangle, or aligned to the left, right, bottom or top. The flag DT_ERASERECT indicates that the rectangle background ought to be filled first with the background color specified. WinDrawText() really just performs WinFillRect() followed by GpiCharStringAt(), having calculated the beginning point according to the alignment flags specified.

When large quantities of text are drawn, spacing between lines and characters becomes an issue. Line spacing must take into account the maximum height

and depth of any character above and below the baseline. Horizontal character spacing is often handled automatically by GpiCharStringAt() or WinDrawText(), but character width must be considered for indenting or aligning columns of text, especially when a proportional font is used. If scroll bars are to be supported, then character width and height are used to define the scrolling units as well. PM provides a data structure of type FONTMETRICS, provides all necessary measurements about the font in use. If the font changes, the font metrics must be queried again to ensure the correct appearance of the application.

FONTMETRICS is a very large data structure, but for the purposes of simple text drawing and line spacing, only a few of its members are significant. Every character is drawn inside a box known as a character cell, and for fixed-width fonts the character cells are all equal in size. For proportional fonts, the cells vary in width. The FONTMETRICS member lAveCharWidth gives the average width of any cell in the font. This value could be used to calculate how far to indent the first line of a paragraph, if necessary. Be careful using it to calculate column spacing in a proportional font. Since this is the average character width, a column of many wide characters may exceed the allowed spacing. In these situations, the value lEmInc would be safer since it defines the width of the cell for the upper-case letter M, which is the widest character in a font.

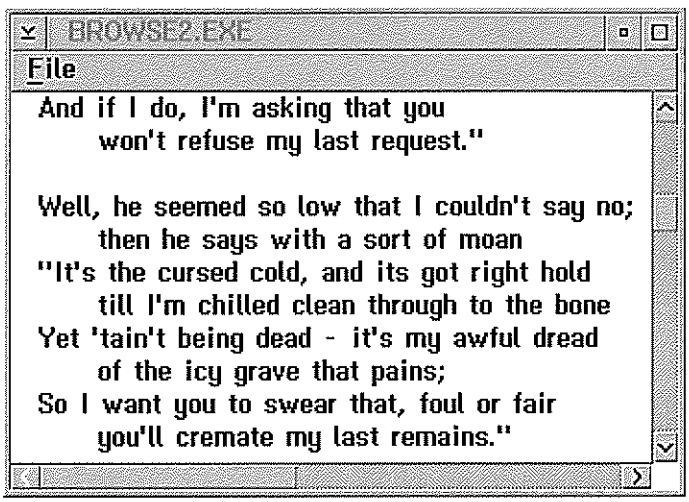


The FONTMETRICS members lMaxAscender and lMaxDescender define, respectively, the maximum height and depth of a character cell above and below the baseline. The sum of these is known as lMaxBaselineExt, the maximum baseline extent. The ascender and descender include a small spacing factor known as lInternalLeading. This is the difference between the maximum ascender and the actual height of the highest character above the baseline. Another spacing factor provided is lExternalLeading, which defines the amount of space allowed between the bottom of one character cell and the top

.of the character cell on the line below. For many fonts, lExeternalLeading is zero because internal leading provides enough vertical spacing between characters.

Good general-purpose spacing can be achieved as follows. The position of the baseline for the first line of text is be calculated by subtracting lMaxAscender from the height of the client window. Then, the vertical spacing is the sum of lMaxBaselineExt and lExternalLeading; this value is subtracted from the position of the previous baseline. If text must be indented, lEmInc may be multiplied by some number of characters to determine the indentation. If text must be scrolled, the horizontal scroll increment should also be lEmInc, and the vertical scroll increment should be the same as the vertical line spacing. Other, more complex algorithms are documented in the on-line PM reference with the FONTMETRICS data structure, but these will suffice for simple text.

Scroll bar support is not difficult for a PM application, but it does require some involved message management. The scroll bar windows do nothing but notify the client window of user scrolling activity; it is up to the client to actually scroll the contents of the window, as well as to send messages back to the scroll bars to update their appearance when scrolling happens. The scroll bars are created by the frame window if the application specifies the flags FCF_VERTSCROLL and FCF_HORZSCROLL among its frame creation flags.

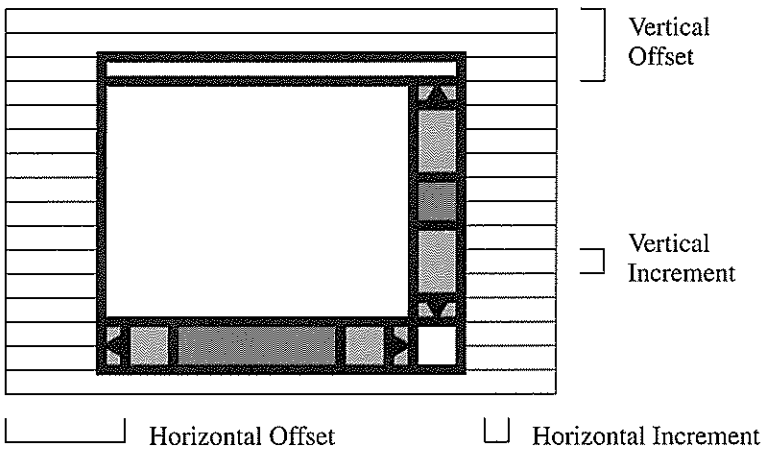


WM_HSCROLL

The message WM_VSCROLL is sent to the client window whenever the user scrolls the vertical scroll bar. The second message parameter contains notification codes, extracted with SHORT2FROMMP(mp2). If the user touches the arrows at either end of the scroll bar, this results in the notification SB_LINEUP or SB_LINEDOWN. Touching the space above or below the

thumb mark results in SB_PAGEUP or SB_PAGEDOWN. Dragging the thumb mark results in SB_SLIDERTRACK. WM_HSCROLL has similar codes for horizontal scrolling. The following discussion will use as an example scrolling lines of text; the scrolling functions provided in the example code browse2.c can be used for general scrolling activity.

Clearly, scrolling is only necessary when the number of lines to be displayed exceeds the height of the window divided by the height of a single line. Define an integer to represent the number of lines that are considered to be above the top of the window. If the first line is displayed, this vertical offset is 0. Another integer is the vertical scrolling increment which is equivalent to the line height. Since the window's height is defined in pixels, dividing the window's height by the vertical increment gives the number of lines that the window can display. When the window paints, it uses the vertical offset to calculate which line is at the top of the window, and draws text from there to the bottom of the window. All vertical scrolling, then, needs to do is adjust the value of the vertical offset.



The original offset can be retrieved from the scroll bar by sending it the SBM_QUERYPOS message. The vertical offset can also be thought of as the distance of the thumb mark from the top of the scroll bar. The offset is returned from the scroll bar window procedure, and returned to the client by WinSendMsg(). If the user scrolls up (SB_LINEUP or SB_PAGEUP) then the vertical offset is reduced, but it cannot be less than zero. For SB_LINEUP, subtract one from the current vertical offset. For SB_PAGEUP, subtract the height of the window divided by the height of a line. If the result is negative, reset it to zero. If the user scrolls down (SB_LINEDOWN or SB_PAGEDOWN) then the vertical offset is increased. For SB_LINEDOWN, add one to the vertical offset. For SB_PAGEDOWN, add the height of the window divided by the height of the line. The vertical offset, however, cannot be greater than the total number of lines less the number of lines in the window. If it was, there would be white space showing at the bottom of the window. If the vertical offset as calculated above exceeds this maximum, reset it to the maxi-

mum. If the user drags the scroll bar slider or thumb mark, the SB_SLIDERTRACK notification is received. In this case, SHORT1FROMMP(mp2) retrieves the new value of the vertical offset.

When the user scrolls, the scroll bar does not reposition the thumb mark. This way, the original offset can be retrieved from the scroll bar first. However, once the new offset has been calculated, the thumb mark must be moved. The application does this by sending the message SBM_SETPOS to the scroll bar, specifying in the first message parameter the new value of the vertical offset. Also, the application must invalidate the client window by calling WinInvalidateRect(), in order to repaint using the new vertical offset value. However, by sending SBM_SETPOS and moving the thumb mark, the application causes the scroll bar to send out another WM_VSCROLL message, this time with the notification SB_SLIDERTRACK. The vertical offset, of course, does not change in this new message. The problem is that if the application invalidates the client window for every WM_VSCROLL message it receives, there may be some unwanted flicker effect as the window gets repainted even when the same lines are displayed. Therefore, the application should only invalidate the window, and only send SBM_SETPOS to the scroll bar, if the vertical offset value has in fact changed.

```
switch (SHORT2FROMMP (mp2))
    {
     case SB_LINEUP:
        sVOffset = max(0, (sVOffset - 1));
        break;
     case SB_LINEDOWN:
        sVOffset = min((sVOffset + 1), (cyData - rclClient.yTop / usVIncr));
        break;
     case SB_PAGEUP:
        sVOffset = max(0, (sVOffset - rclClient.yTop / usVIncr));
        break;
     case SB_PAGEDOWN:
        sVOffset = min((sVOffset + rclClient.yTop / usVIncr),
                       (cyData - rclClient.yTop / usVIncr));
        break;
     case SB_SLIDERTRACK:
        sVOffset = SHORT1FROMMP (mp2);
        break;
     default:
        break;
    }

 if (sVOffset != SHORT1FROMMR (WinSendMsg(hwndVScroll, SBM_QUERYPOS, 0, 0)))
    {
     WinSendMsg(hwndVScroll, SBM_SETPOS, MPFROMSHORT (sVOffset), 0);
     WinInvalidateRect(hwndClient, NULL, FALSE);
    }
```

Horizontal scrolling is similar to vertical scrolling. The horizontal offset is defined to be the number of characters to the left of the left boundary of the window. The horizontal scrolling increment, or column width, is the member lEmInc from the FONTMETRICS data structure. This guarantees that any line, no matter how wide the characters, can be fully displayed and scrolled. The total number of columns to be scrolled is the length in characters of the longest line to be displayed. Exactly the same calculations are performed on the horizontal offset as were done for the vertical offset.

For scrolling to work correctly, however, the scroll bar must be initialized. The scrolling range (number of lines or columns) must be set, and the size of the thumb mark must be proportional to the amount of data displayed. The size and position of the thumb mark, as well as the values of the horizontal and vertical offsets, may change from time to time. This will happen when the window is resized, when new text data must be displayed, or if the font changes. In all these cases, the client window sends two messages to each scroll bar - SBM_SETSCROLLBAR and SBM_SETTHUMBSIZE. In SBM_SETSCROLLBAR, the first message parameter should contain the horizontal or vertical offset so that the scroll bar can position the thumb mark correctly. The second message parameter will contain two short integers that define the scrolling range. Vertical scrolling, for example, would range from zero to the total number of lines, less the number of lines displayed in the window. That way the first line would be displayed at the top of the window, and the last line would be displayed at the bottom of the window with no white space. In SBM_SETTHUMBSIZE, the first message parameter contains two short integers representing the total number of lines and the number of lines the window may display, for vertical scrolling. The scroll bar divides these two to calculate the size of the thumb mark. If, say, only half the total number of lines can currently be displayed in the window, then the thumb mark occupies half of the scroll bar.

```
usHeight = cyWindow / usVIncr;
usWidth = cxWindow / usHIncr;

if (usHeight >= (cyData - *psVOffset) && *psVOffset != 0)
   *psVOffset = max (0, (cyData - usHeight));

if (usWidth >= (cxData - *psHOffset) && *psHOffset != 0)
   *psHOffset = max (0, (cxData - usWidth));

WinSendMsg(hwndVScroll, SBM_SETTHUMBSIZE,
           MPFROM2SHORT (usHeight, cyData), 0);

WinSendMsg(hwndHScroll, SBM_SETTHUMBSIZE,
           MPFROM2SHORT (usWidth, cxData), 0);

WinSendMsg(hwndVScroll, SBM_SETSCROLLBAR,
           MPFROMSHORT (*psVOffset),
           MPFROM2SHORT (0, (cyData - usHeight)));

WinSendMsg(hwndHScroll, SBM_SETSCROLLBAR,
           MPFROMSHORT (*psHOffset),
           MPFROM2SHORT (0, (cxData - usWidth)));
```

The sample program browse2 handles window painting, font management, and scrolling as described above. Painting is restricted to the WM_PAINT message case, and the cache-micro presentation space returned from WinBeginPaint() is used to draw the lines of text. First, WinFillRect() is used to fill the background with the default color. For the sake of simplicity in drawing the text, the entire window rectangle is used for drawing, not just the invalid rectangle. The first line to draw is calculated based on the vertical offset used for scrolling. The horizontal offset is used to calculate the position of the beginning of each line. The vertical position of the first line is calculated by subtracting the maximum ascender of the font from the height of the window. Subsequent lines are drawn by subtracting the sum of the baseline extent and the external leading from the previous vertical position. Finally, only as many lines are drawn as fit in the

current size of the window; any lines above the top or below the bottom are not drawn. The font metrics are queried when the window is created, under the WM_CREATE message case. The necessary values are saved for use in the WM_PAINT message case.

When the window is resized, the scroll bars are reset by one call to the application function fnFixScrollSize(). The scroll bar window handles, height and width of the data (number of lines and maximum line length), height and width of the window, and the scroll increments are passed by value. The current vertical and horizontal offsets are passed by reference, since this function may update them. In the function fnFixScrollSize(), the new number of lines and columns supported by the new window size are calculated as usHeight and usWidth. Then new values of the vertical and horizontal offsets are calculated. When a window is resized, as much as possible of the same data should be displayed starting from the top left corner. Thus, only if the window has been scrolled all the way to the bottom or all the way to the right are new offsets calculated, for then more data can be displayed at the top or left if the window is made bigger. Finally, the messages SBM_SETTHUMBSIZE and SBM_SETSCROLLBAR are sent to each scroll bar to set the new proportional values.

When scrolling occurs, the message cases WM_VSCROLL and WM_HSCROLL are invoked in the client window procedure. The functions fnVScroll() and fnHScroll() are invoked to perform the scroll bar management. These functions use a switch...case construction to determine which notification code was sent, and calculate the new offsets accordingly. Only if the thumb mark has actually moved, is the scroll bar updated and the client window repainted.

The message WM_CHAR is handled by the client window only for scrolling purposes. This invokes the application function fnCharScroll() which determines if the arrow keys, page up, or page down were struck. If so, the WM_CHAR message is simply passed on to the appropriate scroll bar, since the scroll bars can handle character messages automatically and send out the appropriate WM_VSCROLL or WM_HSCROLL messages.

The message UM_FILEREAD is received from the secondary thread when the file has been successfully loaded from disk. Then, the client window procedure extracts the number of lines, and the length of the longest line, from the message parameters and stores these as the new height and width of data. A call to the function fnFixScrollSize() will ensure that new scrolling proportions are calculated for the new file.

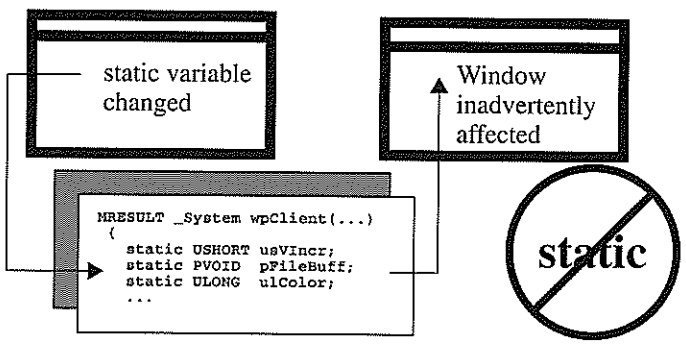
The message WM_PRESPARAMCHANGED is received from PM when the user changes font for the window. This can be done by dragging a new font from the font palette in the system setup folder. In this message case, new font metrics must be queried. Here, as in WM_CREATE, a temporary cache-micro presentation space is obtained with the WinGetPS() API function. Then,

GpiQueryFontMetrics() retrieves the FONTMETRICS data structures, and new spacing values are calculated. WinReleasePS() returns the presentation space. A call to fnFixScrollSize() updates the scrolling proportions to correspond to the new font size (the horizontal and vertical scroll increments are different, as are the number of lines and columns supported by the window.) Finally, WinInvalidateRect() forces the window to be repainted with the new font.

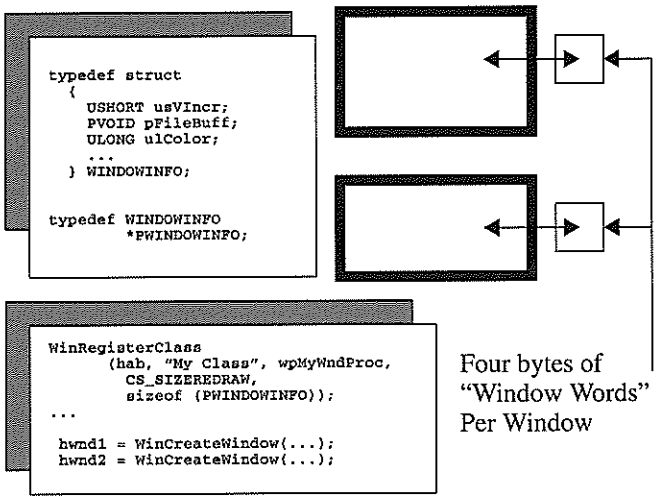
Window Data Encapsulation

Every message in PM ultimately results in a function call being made to a window procedure. In the window procedure, exactly one case statement is entered and the message is processed. Then, the next message results in a new invocation of the window procedure. This raises a question of how data might be retained by the window procedure for use across multiple message cases. In the example program browse2, the offset and increment values for scrolling and line spacing are used in the WM_PAINT, WM_VSCROLL, and WM_HSCROLL message cases and updated in the WM_SIZE, WM_PRESPARAMCHANGED, and UM_FILEREAD message cases.

In normal C language functions, variables of the static storage class are often used to retain information. Static variables are not stack-allocated, and retain their value through subsequent invocations of the same function. Static variables are very dangerous in PM window procedures. Remember that the window procedure is registered with the window class. This means that the same window procedure will be used to service messages for all windows created of a given class. Static variables will then retain their value, not just across different message cases for the same window, but in fact across multiple windows, causing unwanted side effects to occur. Consider what would happen if the sample program browse2 were to create multiple text-viewing windows. The user could change font or change file for each window separately. If the offset and increment values were kept as static variables, then a font or file change in one window would change scrolling parameters across all windows, which would be unacceptable.



Instead, PM provides a way to encapsulate such data on a per-window basis specifically to retain information that must be persistent through the life of the window. The technique for this data encapsulation begins with the WinRegisterClass() API function. When a class is registered, the last parameter of WinRegisterClass() defines a number of bytes of storage. This is an amount of memory that will be allocated by PM whenever a window of this class is created. Any amount of storage may be requested, but it is easiest to use just four bytes, or the size of a pointer. This storage is known as window words, and can later be accessed using several PM API functions, most commonly WinSetWindowPtr() and WinQueryWindowPtr().

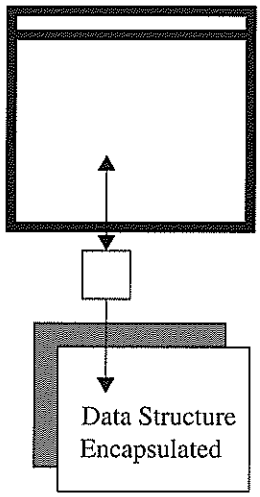


When the window is created, PM allocates the four bytes of window words storage. In the WM_CREATE message case, the window procedure would then dynamically allocate and initialize a user-defined data structure that holds all the instance data that the window needs to retain. Since the structure is dynamically allocated, a new one is set up for each window created. The pointer to the data structure can be stored in the window words by calling WinSetWindowPtr(). This function accepts as its first parameter the window handle. The third parameter is the pointer, and the second parameter is an offset defining where the pointer should start in the window words. Since exactly four bytes, the size of a pointer, had been requested, the offset is normally set to zero.

In other message cases, where the instance data must be retrieved to be modified or used, the WinQueryWindowPtr() API function returns the pointer from the window words. The two parameters to WinQueryWindowPtr() are the window handle and the offset, again usually zero. In this way, each of multiple windows could retain its own scrolling information. When the data structure is allocated, modified, or read, it is always distinguished by the window handle. A different data structure is maintained for each window. There are no side effects propagated if one window changes font, or another window changes file. With this method of data encapsulation available, it is good style to avoid

static variables entirely in the window procedure, and store any persistent information in an instance data structure.

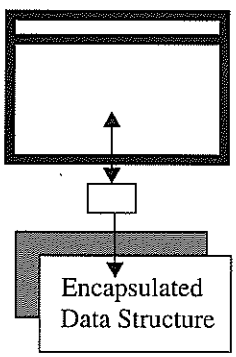
```
MRESULT _System wpClient(...)
{
  MRESULT mr = 0;
  PWINDOWINFO pwi;

 switch (msg)
   {
     case WM_CREATE:
        pwi = (PWINDOWINFO)
           malloc(sizeof(WINDOWINFO));
        pwi -> usVIncr = 0;
         ...
        WinSetWindowPtr(hwnd, 0, pwi);
        break;

     case WM_PAINT:
        pwi = WinQueryWindowPtr(hwnd, 0);
        ...
        break;
      ...
     case WM_DESTROY:
        pwi = WinQueryWindowPtr(hwnd, 0);
        free(pwi);
        break;
     ...
```



The instance data technique can be combined with WinCreateWindow() to provide a powerful method of window initialization. The main routine that is creating instance windows of a class may want to create each window a little differently — a different color in one, or a different initial file to display in another. In this case, the main routine could allocate and initialize a separate data structure for each window being created. The last two parameters of WinCreateWindow() are defined as pointers to control data and presentation parameter structures for control windows. These pointers become the message parameters mp1 and mp2, respectively, in WM_CREATE. For an application-defined client window of a private window class, control data and presentation parameters are undefined. These pointer parameters could then be used by the application to pass a structure of instance data to any window, upon creation.

```
pwi = (PWINDOWINFO) malloc(sizeof(WINDOWINFO));

pwi -> ulColor = ...;

hwndClient = WinCreateWindow(...,
                             pwi,    // Initialize window
                             NULL);  // No Pres. Parms.
```

```
PWINDOWINFO pwi;
...
   case WM_CREATE:
      pwi = PVOIDFROMMP (mp1);
        WinSetWindowPtr(hwnd, 0, pwi);
```



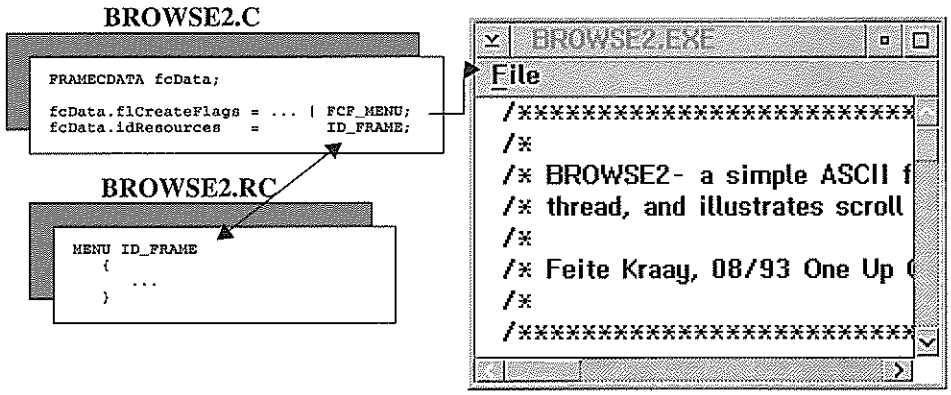
The sample program browse2 uses instance data in order to avoid static variables. In the header file browse2.h, there is a typedef statement defining a data structure that will hold all the instance data. This user-defined structure is named TEXTINFO. In the source file browse2.c, WinRegisterClass() asks for four bytes of window words to be allocated per window. In the WM_CREATE

.message case, the window procedure uses malloc() to allocate storage for the TEXTINFO instance data structure, which is then initialized with scrolling values based on the default font for the window. All other message cases call WinQueryWindowPtr() first to retrieve the pointer to the data structure, and then use or modify the contents of the structure. Finally, in the WM_DESTROY message case, when the window is being destroyed, the window procedure calls free() to release the memory occupied by the instance data structure.

Menu Resource Management

The menu, or action bar, provides the basis for most of the user interaction with a PM application. The *Common User Access Advanced Interface Design Guide* supplies recommendations regarding the appearance of the menu, and the options supplied. The concern here is how the menu is defined and used by the application — what API functions and what messages are important. The menu is not difficult to deal with. Some of its actions have already been described in previous sections.

The menu is one instance of a resource in Presentation Manager, and it is defined in a separate resource file. Other examples of resources are icons, pointers, accelerator keys and dialog boxes. Resources in general are all language-specific or interface-specific objects in a PM application. These are all defined in a separate resource source file which is compiled and added to the application with the resource compiler, RC.EXE. This way, the resources may be maintained independently of the application's source code.



Although the menu could be created dynamically by means of API functions, it is normally created statically in the resource file. Then, turning on the flag FCF_MENU in the frame creation flags for WinCreateStdWindow() or WinCreateWindow() will direct the frame window to locate the menu resource and create the menu control. The menu may be modified later by sending it messages from the client window. This would be used to enable or disable menu items, or to add or remove menu items.

The menu template is defined in the resource file using the keyword MENU. A pair of braces {...} delimits the menu; the keywords BEGIN and END will also do. Within the menu, the keyword SUBMENU defines each choice that will be displayed across the top of the menu. Typical submenus are File, Edit, or Help. Braces or BEGIN and END delimit the submenu. Within each submenu, the keyword MENUITEM defines each individual selectable choice such as Open, Save, Cut, or Copy.

BROWSE2.RC

```
#include <os2.h>
#include "browse2.h"

MENU ID_FRAME
  {
   SUBMENU "~File", IDM_FILE
      {
       MENUITEM "~Open...", IDM_OPEN
       MENUITEM "~Clear", IDM_CLEAR
       MENUITEM SEPARATOR
       MENUITEM "E~xit...", IDM_EXIT
      }
  }
```

BROWSE2.H

```
...
#define ID_FRAME  100

#define IDM_FILE  300
#define IDM_OPEN  301
#define IDM_CLEAR 302
#define IDM_EXIT  303
...
```

For each submenu and menu item, the text must be specified in double quotation marks. This text is drawn in the appropriate location within the menu. The tilde character (~) identifies which letter is used as a mnemonic for each submenu or menu item. When the user has assigned focus to the action bar (by pressing F10, for example), then pressing the mnemonic key opens the associated submenu or selects the menu item. The mnemonic characters must be unique across all submenus, and unique across all menu items within any submenu. One tab character (\t) may be inserted in the text; this is useful for aligning any accelerator key values to the right of the menu text. The "..."(called an ellipsis) following certain menu items is a CUA convention indicating that selection of the menu will result in a dialog or message box being displayed. The keyword SEPARATOR on a menu item causes a horizontal line to be drawn between items. This is useful for grouping related choices, or isolating a choice such as Exit... by itself.

The menu, and all submenus and menu items, must have numeric ID values. These should all be constants defined in the application's header file. The menu ID must also be passed to the frame window by means of the idResources member of the FRAMECDATA structure (for WinCreateWindow()) or the frame window ID in the second-last parameter of WinCreateStdWindow(). If the same menu ID value is not used in the resource file and WinCreateWindow() or WinCreateStdWindow(), then the frame window creation will fail since the menu cannot be found. The ID values for the submenus and menu items should be defined as distinct constants in the header file. Then, the application will always be able to distinguish one menu command from another.

Style flags may optionally be specified for submenus and menu items. Most common is the flag MIS_TEXT, indicating that the submenu or menu item is to be drawn as text, although MIS_BITMAP could be used to add graphics to a menu. MIS_TEXT is the default, and need not be specified explicitly.

```
MENU ID_FRAME
     {
       SUBMENU "~File", IDM_FILE
             {
               MENUITEM "Item ~1", IDM_ONE, MIS_TEXT
               MENUITEM "Item ~2", IDM_TWO,    , MIA_CHECKED
               MENUITEM "Item ~3", IDM_THREE, , MIA_DISABLED
             }
          ...
     }
```

```
/* Client window procedure */

HWND hwndFrame, hwndMenu;

hwndFrame = WinQueryWindow(hwnd, QW_PARENT);
hwndMenu = WinWindowFromID(hwndFrame, FID_MENU);

WinEnableMenuItem(hwndMenu, IDM_THREE, TRUE); // Enable Item 3
WinEnableMenuItem(hwndMenu, IDM_TWO, FALSE);  // Disable Item 2
```

Menu item attributes can be specified following the style flags. The styles and attributes are delimited by commas, so if no styles are specified then a comma must still be left before the attributes are specified. The attribute MIA_DISABLED can be used to disable a menu item or submenu so that it cannot be selected by the user. Menu items may be enabled and disabled later using the macro API WinEnableMenuItem(). The attribute MIA_CHECKED is used to draw a check mark next to a menu item. Checked menu items are normally used to toggle options that can be set for a window. An editor, for example, could use checked menu items for user configurations such as auto save or word wrap. Menu items may be checked or unchecked later using the macro API WinCheckMenuItem().

WinEnableMenuItem() and WinCheckMenuItem() really send the message MM_SETITEMATTR to the menu window, to change the respective attributes. Each of these macros accepts three parameters, beginning with the menu window handle. This handle can be found in a two-step process. The menu is a child of the frame, so WinWindowFromID() using the value FID_MENU returns the menu window handle if the frame window handle is known. The frame is the parent of the client, and its handle can be determined using WinQueryWindow(), specifying the relationship flag QW_PARENT. The second parameter to WinEnableMenuItem() or WinCheckMenuItem() is the ID value of the item to be affected, as defined in the header file. The third parameter to these macros is a boolean flag, TRUE to check or enable, FALSE to remove the check or disable.

As long as the menu has been defined, and the resource compiler has successfully compiled and embedded the resources, the user may now select options from the application's menu. Whenever the user makes a selection, the WM_COMMAND message is posted from the menu window to the frame, and sent from the frame to the client window. The client window procedure examines the low end of the first message parameter using SHORT1FROMMP (mp1). This contains the ID number of the selected menu item. Usually, a

nested switch...case construction is used to take action in response to the different possible menu item selections. The second parameter of WM_COMMAND contains additional information such as whether the selection was made with mouse or keyboard, and whether the message was generated from the menu, the accelerator table, or a push button.

```
MRESULT _System wpClient (HWND hwnd, ULONG msg,
                          MPARAM mp1, MPARAM mp2)
 {
   switch (msg)
       {
         ...
         case WM_COMMAND:

              switch (SHORT1FROMMP (mp1)
                   {
                     case IDM_OPEN:     // Do file open
                         break;
                     case IDM_CLEAR:    // Clear window
                         break;
                     case IDM_EXIT:     // Do exit processing
                         break;
                     default:
                         break;
                   }
              break;
        ...
       }
 }
```

The sample program browse2 contains a menu defined in the resource file browse2.rc. The make file browse2.mak contains the commands rc -r browse2.rc to generate the compiled resource in the file browse2.res. This file is then embedded into the program by the command RC BROWSE2.RES BROWSE2.EXE. The resource file browse2.rc includes the header file os2.h so that the keywords are recognized.

It also includes the application header browse2.h so that the menu, submenu, and menu item IDs can be set up as constants. Then the menu is built as described above, with only one submenu and three menu items. The keywords MENUITEM SEPARATOR draw a horizontal line to separate two unrelated menu items.

In the source file browse2.c, the WM_COMMAND message case handles all the menu interaction. IDM_OPEN represents the choice Open under File, and here the user is asking to select a new file. The API function WinFileDlg() is used to display a standard file selection dialog box, customized by the data structure of type FILEDLG. This API returns the selected file name in the member szFullFile of the FILEDLG structure. The file name and client window handle are copied into a data structure of type OPENINFO, which is defined in the header file browse2.h. The address of this structure is passed as a parameter to the thread function fnOpenThread, which is started using DosCreateThread(). When the thread finishes, it posts the user-defined message UM_FILEREAD to the client window and the file is then displayed.

IDM_CLEAR represents the option Clear under File on the menu, and is used to indicate that the current file should no longer be displayed. The file buffer is released, scrolling parameters are reset to zero, and the window is invalidated so that it repaints itself with just the background color.

IDM_EXIT represents the option Exit under File, and is selected when the user wishes to terminate the application. In this case, the client window just posts WM_CLOSE to itself, in order to invoke the message box prompt for confirmation to end. If the user answers yes to the message box, the window procedure posts WM_QUIT to terminate.

Summary

- Where To Go Next
- OS/2 Awareness Series
- About the Authors

Where to go Next

You should now have a better understanding of, and appreciation for, the capability and power of OS/2 version 2.1 as a development system. If you want to learn more, there are several things to do.

If you are interested in communications, a good place to start is the next volume in the *One Up OS/2 Awareness Series* which is described in more detail below. If you just want to get a more detailed understanding of how to use the OS/2 system, visit any good bookstore. You will find several excellent books there covering topics ranging from how to set up your system to deep technical discussions on how the various components of OS/2 were designed.

OS/2 Awareness Series

Volume 1 increases your understanding of how OS/2 can make your computing time more productive and pleasant. It includes descriptions of the base system and the productivity applets that are included with the system. This book also includes hints on how to maximize the performance of the DOS emulation capabilities of OS/2 and how to install and use the multimedia support.

This volume, Volume 2, was designed to increase your understanding of how to write programs for OS/2. You should have a clear idea of the programming concepts in OS/2 version 2.1 and be able to use the sample programs as the basis for programs of your own design.

If you want more detailed programming instruction, you should contact One Up for a class schedule. There are several programming courses available that cover all aspects of OS/2 programming.

Volume 3 in the *One Up OS/2 Awareness Series* looks at OS/2 as a communications platform. It includes an introduction to LAN Server version 3 and Communications Manager/2. Installation instructions are included as well as descriptions of the administrative tasks necessary to build or connect to a LAN.

About the Authors

Feite Kraay spent three years at IBM Canada Education developing and teaching courses on OS/2 and Presentation Manager programming, from version 1.1 through 2.0. In early 1992 he joined One Up Corporation, continuing his OS/2 education and consulting activities. He is now the general manager of One Up Computer Services Ltd., the Canadian subsidiary of One Up Corporation. Feite Kraay holds a Bachelor of Mathematics degree from the University of Waterloo

Larry Pollis spent 10 years with Rolm. From there, he joined IBM working in the OS/2 technical support group. He wrote several technical bulletins and presented several topics on the IBM Field Television Network about OS/2.

Craig Chambers is a graduate of Purdue University with BS and MS degrees. He worked for IBM for 23 years in various positions including; large account System Engineer, 3270 and 3270-PC technical support, and PC-DOS and OS/2 technical support. During this time, he wrote several articles for various technical journals, made presentations at COMMON, and participated in several IBM Field Television Network broadcasts.

Appendix A - BROWSE1

```
/*************************************************************************/
/*                                                                       */
/* BROWSE1- a simple ASCII file browser - loads the file in a secondary  */
/* thread, and prints the file to stdout.                                */
/*                                                                       */
/* Feite Kraay, 08/93 One Up Corporation.                                */
/*                                                                       */
/*************************************************************************/

/*************************************************************************/
/*                                                                       */
/* Header file includes:                                                 */
/*                                                                       */
/*************************************************************************/

#define INCL_DOSSEMAPHORES // OS/2 Semaphores
#define INCL_DOSPROCESS    // OS/2 Tasking
#define INCL_DOSMEMMGR     // OS/2 Memory Handling
#define INCL_DOSFILEMGR    // OS/2 File Functions

#include <os2.h>           // Include OS/2 Headers
#include <stdio.h>         // Include C I/O library
#include <string.h>        // Include C string function library
#include <stdlib.h>        // Include C standard library
#include "browse1.h"       // Application header

/*************************************************************************/
/*                                                                       */
/* Functon main - primary thread entry point. Start the thread to load the */
/* file, and display message. Then print the file to stdout when the      */
/* thread finishes.                                                      */
/*                                                                       */
/*************************************************************************/

int main(int argc, char *argv[])
{
 HEV       hevLoad;
 POPENINFO pOpenInfo;
 TID       tidOpen;
 PCH       pWork;

 /************************************************************************/
 /*                                                                      */
 /* As long as there is one argument, accept it as a file name and kick  */
 /* off the thread to load that file. Use an event semaphore to know when */
 /* the thread has finished.                                             */
 /*                                                                      */
 /************************************************************************/
```

```
 if (argc == 2)
    {
     pOpenInfo = (POPENINFO) malloc (sizeof (OPENINFO));

     DosCreateEventSem (NULL, &hevLoad, 0, FALSE);

     strcpy (pOpenInfo -> szFileName, argv[1]);
     pOpenInfo -> hevLoad = hevLoad;

     DosCreateThread (&tidOpen, (PFNTHREAD) fnOpenThread,
                      (ULONG) pOpenInfo, 0L, 8192);
    }
 else
    {
     printf ("Usage is ==> browse1 <filename>\n");
     DosExit (EXIT_PROCESS, 0);
    }

 /*************************************************************************/
 /*                                                                       */
 /* After the thread has started, main could be doing other work. Just    */
 /* print a couple of messages to the user, and wait on the semaphore to  */
 /* know when to start printing the file.                                 */
 /*                                                                       */
 /*************************************************************************/

 printf ("The thread is now loading the file ...\n");
 printf ("The main function could be doing other useful work ...\n");
 printf ("We'll just wait on the thread to finish.\n\n");

 DosWaitEventSem (hevLoad, -1);

 /*************************************************************************/
 /*                                                                       */
 /* The thread notifies main of an error by specifying a 0 file size -    */
 /* then just print an error message. Otherwise, just use putc to print   */
 /* the file contents to stdout. Free the memory when the file has been   */
 /* printed.                                                              */
 /*                                                                       */
 /*************************************************************************/

 if (pOpenInfo -> ulSize == 0)
    {
     printf ("An error occurred loading the file\n");
     DosCloseEventSem (hevLoad);
     free (pOpenInfo);
     DosExit (EXIT_PROCESS, 0);
    }
 else
    {
     printf ("File successfully loaded ...\n\n");
     for (pWork = pOpenInfo -> pchFile;
             pWork < pOpenInfo -> pchFile + pOpenInfo -> ulSize;
                pWork ++)
        {
         putc (*pWork, stdout);
        }
     DosFreeMem (pOpenInfo -> pchFile);
    }

 /*************************************************************************/
 /*                                                                       */
 /* Clean up and terminate - free the data structure used to pass info    */
 /* between main and the thread, and close the semaphore.                 */
 /*                                                                       */
 /*************************************************************************/

 free (pOpenInfo);
 DosCloseEventSem (hevLoad);
 DosExit (EXIT_PROCESS, 0);
}

/*************************************************************************/
/*                                                                       */
/* Thread function fnOpenThread - open the file, allocate memory, read it, */
/* and post a semaphore to indicate completion.                          */
/*                                                                       */
/*************************************************************************/

VOID _System fnOpenThread (POPENINFO pOpenInfo)
{
```

```
HFILE             hFile;
ULONG             ulAction;
FILESTATUS        fstsFile;
ULONG             ulBytes;
BOOL              boolError;

boolError = FALSE;

do {
    /*************************************************************************/
    /*                                                                       */
    /* Open the file, privately to this process. Open for read only, and     */
    /* fail if the file does not exist.                                      */
    /*                                                                       */
    /*************************************************************************/

    if (DosOpen (pOpenInfo -> szFileName, &hFile, &ulAction, 0, 0,
                 OPEN_ACTION_FAIL_IF_NEW |
                 OPEN_ACTION_OPEN_IF_EXISTS,
                 OPEN_SHARE_DENYWRITE |
                 OPEN_ACCESS_READONLY,
                 0))
       {
        boolError = TRUE;
        break;
       }

    /*************************************************************************/
    /*                                                                       */
    /* Determine the size of the file, from a FILESTATUS data structure.     */
    /*                                                                       */
    /*************************************************************************/

    if (DosQueryFileInfo (hFile, FIL_STANDARD, &fstsFile,
                          sizeof (FILESTATUS)))
       {
        boolError = TRUE;
        break;
       }

    /*************************************************************************/
    /*                                                                       */
    /* Allocate sufficient memory to hold the contents of the file.          */
    /*                                                                       */
    /*************************************************************************/

    if (DosAllocMem ((PPVOID) &(pOpenInfo -> pchFile), fstsFile.cbFileAlloc,
                     PAG_WRITE | PAG_COMMIT))
       {
        boolError = TRUE;
        break;
       }

    /*************************************************************************/
    /*                                                                       */
    /* Read the file into the memory buffer just allocated.                  */
    /*                                                                       */
    /*************************************************************************/

    if (DosRead (hFile, (PVOID) pOpenInfo -> pchFile, fstsFile.cbFile,
                 &ulBytes))
       {
        boolError = TRUE;
        DosFreeMem (pOpenInfo -> pchFile);
        break;
       }

   } while (FALSE);

/*****************************************************************************/
/*                                                                           */
/* Close the file when finished. If an error occurred at any step, set a     */
/* value of 0 in the shared data structure to indicate failure. Otherwise    */
/* return the file pointer and size in the data structure, to indicate       */
/* success and allow main to print the file.                                 */
/*                                                                           */
/*****************************************************************************/

DosClose (hFile);

if (boolError)
```

```
    {
     pOpenInfo -> pchFile = NULL;
     pOpenInfo -> ulSize = 0;
    }
  else
    {
     pOpenInfo -> ulSize = fstsFile.cbFile;
    }

  DosPostEventSem (pOpenInfo -> hevLoad);

  DosExit (EXIT_THREAD, 0);
}
```

The following is the MAKE file for browse1.c:

BROWSE1.MAK

```
browse1.exe: browse1.obj
        link386 /NOI /PM:VIO /DEBUG browse1.obj;

browse1.obj: browse1.c browse1.h
        icc /c /Ss /Ti+ /Kb /Gm browse1.c
```

The following is the header file, browse1.h, for browse1:

BROWSE1.H

```
/*****************************************************************************/
/*                                                                           */
/* Text browser demo - application header.                                   */
/*                                                                           */
/*****************************************************************************/

typedef struct                // Thread argument data structure
   {
    CHAR  szFileName [255];   // File name
    HEV   hevLoad;            // Semaphore handle
    PCH   pchFile;            // Pointer to file buffer
    ULONG ulSize;             // Size of file
   } OPENINFO, *POPENINFO;

// Function prototype

VOID _System fnOpenThread (POPENINFO pOpenInfo);
```

Appendix B - BROWSE2

```
/*****************************************************************************/
/*                                                                           */
/* BROWSE2- a simple ASCII file browser - loads the file in a secondary      */
/* thread, and illustrates scroll bar support.                               */
/*                                                                           */
/* Feite Kraay, 08/93 One Up Corporation.                                    */
/*                                                                           */
/*****************************************************************************/

/*****************************************************************************/
/*                                                                           */
/* Header file includes:                                                     */
/*                                                                           */
/*****************************************************************************/

#define INCL_WIN          // PM Win APIs
#define INCL_GPI          // PM Graphics APIs
#define INCL_DOSPROCESS   // OS/2 Tasking
#define INCL_DOSMEMMGR    // OS/2 Memory Handling
#define INCL_DOSFILEMGR   // OS/2 File Functions

#include <os2.h>          // Include OS/2 Headers
#include <string.h>       // Include C string functions
#include <stdlib.h>       // Include C standard library
#include "browse2.h"      // Application header

/*****************************************************************************/
/*                                                                           */
/* Function main - primary thread entry point. Window creation, message      */
/* loop and termination.                                                     */
/*                                                                           */
/*****************************************************************************/

int main(int argc, char *argv[])
{
 HAB         hab;
 HMQ         hmq;
 QMSG        qmsg;
 HWND        hwndClient, hwndFrame;
 ULONG       flCreateFlags;
 POPENINFO   pOpenInfo;
 TID         tidOpen;

 /*****************************************************************************/
 /*                                                                           */
 /* Create anchor block and initialize this thread to PM.                     */
 /* Create message queue on the anchor block - use default size               */
 /* Register the class for the client window - the window procedure is        */
 /* defined below, and 4 bytes of additional storage will be allocated by     */
```

```
   /* PM upon creation of any window of this class.                        */
   /*                                                                      */
   /************************************************************************/

   hab = WinInitialize (0);

   hmq = WinCreateMsgQueue (hab, 0);

   WinRegisterClass (hab, CLIENT_CLASS_NAME, wpClientWndProc,
                     CS_SIZEREDRAW, 4);

   /************************************************************************/
   /*                                                                      */
   /* Initialize the frame creation flags that will customize the appearance */
   /* and behaviour of the frame window assembly. Then create the frame and  */
   /* client windows in one step, using WinCreateStdWindow. The identifier   */
   /* ID_FRAME is used to locate the menu resource requested.              */
   /*                                                                      */
   /************************************************************************/

   flCreateFlags = FCF_TITLEBAR       |      // Title Bar
                   FCF_SYSMENU        |      // System Menu Icon
                   FCF_SIZEBORDER     |      // Normal Sizeable Border
                   FCF_MINMAX         |      // Minimize/Maximize
                   FCF_MENU           |      // Action Bar
                   FCF_VERTSCROLL     |      // Vertical Scroll Bar
                   FCF_HORZSCROLL     |      // Horizontal Scroll Bar
                   FCF_SHELLPOSITION  |      // PM Default Size/Position
                   FCF_TASKLIST;             // Windows List Entry

   hwndFrame = WinCreateStdWindow (HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
                                   CLIENT_CLASS_NAME, NULL, 0, 0, ID_FRAME,
                                   &hwndClient);

   /************************************************************************/
   /*                                                                      */
   /* As long as there is one argument, accept it as a file name and kick  */
   /* off the thread to load that file.                                    */
   /*                                                                      */
   /************************************************************************/

   if (argc == 2)
      {
       pOpenInfo = (POPENINFO) malloc (sizeof (OPENINFO));

       strcpy (pOpenInfo -> szFileName, argv[1]);

       pOpenInfo -> hwnd = hwndClient;

       DosCreateThread (&tidOpen, (PFNTHREAD) fnOpenThread,
       (ULONG) pOpenInfo, 0L, 8192);
      }

   /************************************************************************/
   /*                                                                      */
   /* Enter the message loop - retrieve one message at a time from the input */
   /* queue, and dispatch it synchronously to whatever window procedure    */
   /* ought to deal with it.                                               */
   /*                                                                      */
   /************************************************************************/

   while (WinGetMsg (hab, &qmsg, 0, 0, 0))
      WinDispatchMsg (hab, &qmsg);

   /************************************************************************/
   /*                                                                      */
   /* WM_QUIT causes WinGetMsg to return FALSE, terminating the message loop */
   /* and therefore the application. Destroy the frame window (and therefore */
   /* all its descendants), destroy the message queue and the anchor block.  */
   /*                                                                      */
   /************************************************************************/

   WinDestroyWindow (hwndFrame);

   WinDestroyMsgQueue (hmq);

   WinTerminate (hab);

   DosExit (EXIT_PROCESS, 0);
}
```

```
/*************************************************************************/
/*                                                                       */
/* Client window procedure wpClientWndProc - handle all messages on behalf */
/* of this application's main window.                                    */
/*                                                                       */
/*************************************************************************/

MRESULT wpClientWndProc (HWND hwnd, ULONG msg, MPARAM mp1, MPARAM mp2)
{
 MRESULT     mr = (MRESULT) 0;
 PTEXTINFO pti;

 switch (msg)
    {
     case WM_CREATE:

     /*********************************************************************/
     /*                                                                   */
     /* Window creation - allocate and initialize a window instance data  */
     /* structure, whose pointer will be stored at offset 0 in the 4 bytes */
     /* of window words data asked for by WinRegisterClass.               */
     /*                                                                   */
     /*********************************************************************/
         {
          HPS hps;
          FONTMETRICS fm;

          pti = (PTEXTINFO) malloc (sizeof (TEXTINFO));
          WinSetWindowPtr (hwnd, 0, pti);

          // Scroll bar window handles

          pti -> hwndVS = WinWindowFromID (WinQueryWindow (hwnd, QW_PARENT),
                                           FID_VERTSCROLL);
          pti -> hwndHS = WinWindowFromID (WinQueryWindow (hwnd, QW_PARENT),
                                           FID_HORZSCROLL);

          hps = WinGetPS (hwnd);
          GpiQueryFontMetrics (hps, sizeof (FONTMETRICS), &fm);
          pti -> usVIncr = fm.lMaxBaselineExt  // Height of a line of text
                         + fm.lExternalLeading;
          pti -> usHIncr = fm.lEmInc;          // Maximum character width
          pti -> usTopIncr = fm.lMaxAscender;  // Offset to draw from top
          pti -> sVOffset = 0;                 // Positioning of text
          pti -> sHOffset = 0;
          WinReleasePS (hps);

          pti -> ulHeight = 0;                 // Number of lines of text
          pti -> ulWidth = 0;                  // Max. characters per line

          pti -> boolText = FALSE;             // No text initially drawn
         }
         break;

     case WM_SIZE:

     /*********************************************************************/
     /*                                                                   */
     /* When the window is re-sized, we need to adjust the proportional   */
     /* size of the scroll bar thumb mark; this is done by the function   */
     /* fnFixScrollSize below.                                            */
     /*                                                                   */
     /*********************************************************************/

        pti = WinQueryWindowPtr (hwnd, 0);

        fnFixScrollSize (pti -> hwndVS, pti -> hwndHS,
                         pti -> ulWidth, pti -> ulHeight,
                         SHORT1FROMMP (mp2), SHORT2FROMMP (mp2),
                         pti -> usVIncr, pti -> usHIncr,
                         &(pti -> sVOffset), &(pti -> sHOffset));
        break;

     case WM_PRESPARAMCHANGED:

     /*********************************************************************/
     /*                                                                   */
     /* User has dragged a new font to this window, from the font palette. */
     /* We need to record the new line height, character width, and offset */
     /* for drawing. This also affects the proportional size of the scroll */
     /* bar thumb mark, so call fnFixScrollSize again. Then repaint in the */
```

```
        /* new font.                                                          */
        /*                                                                    */
        /**********************************************************************/
           {
            HPS hps;
            FONTMETRICS fm;
            RECTL rcl;

            pti = WinQueryWindowPtr (hwnd, 0);

            hps = WinGetPS (hwnd);
            GpiQueryFontMetrics (hps, sizeof (FONTMETRICS), &fm);
            pti -> usVIncr = fm.lMaxBaselineExt + fm.lExternalLeading;
            pti -> usHIncr = fm.lEmInc;
            pti -> usTopIncr = fm.lMaxAscender;
            WinReleasePS (hps);

            WinQueryWindowRect (hwnd, &rcl);

            fnFixScrollSize (pti -> hwndVS, pti -> hwndHS,
                             pti -> ulWidth, pti -> ulHeight,
                             rcl.xRight, rcl.yTop,
                             pti -> usVIncr, pti -> usHIncr,
                             &(pti -> sVOffset), &(pti -> sHOffset));

           WinInvalidateRect (hwnd, NULL, 0);
           }
           break;

        case WM_PAINT:

        /**********************************************************************/
        /*                                                                    */
        /* Window repainting. If there is no text to draw, just fill with     */
        /* background colour. If there is text, the file has been prepared    */
        /* as a sequence of zero-terminated strings. Step through the file    */
        /* until we get to the string that should be at the top of the        */
        /* window, then draw only the strings from there to the bottom of the */
        /* window.                                                            */
        /*                                                                    */
        /**********************************************************************/
           {
            HPS         hps;
            RECTL       rectlClient;
            POINTL      ptlText;
            ULONG       i;
            PCH         pWork;
            ULONG       ulLength;
            ULONG       ulBegin, ulEnd;

            pti = WinQueryWindowPtr (hwnd, 0);

           /*******************************************************************/
            /*                                                                */
            /* Determine the dimension of the client window, then get a       */
            /* presentation space and paint the background of the window.     */
            /*                                                                */
           /*******************************************************************/

            WinQueryWindowRect (hwnd, &rectlClient);
            hps = WinBeginPaint (hwnd, 0, NULL);
            WinFillRect (hps, &rectlClient, SYSCLR_WINDOW);

           /*******************************************************************/
            /*                                                                */
            /* If there is text, get the pointer to the buffer. Drawing begins */
            /* in the top-left corner of the window. The variable ulBegin, or  */
            /* the vertical offset, is equivalent to the number of the first   */
            /* line in the window (or the 'height' above the top of the window */
            /* where the text would begin.) Step through the file until that   */
            /* line, and then draw from there to the bottom of the window      */
            /* only.                                                          */
            /*                                                                */
           /*******************************************************************/

            if (pti -> boolText)
               {
                pWork = (PCH) pti -> pFile;
                ptlText.x =  pti -> usHIncr - pti -> usHIncr * pti -> sHOffset;
                ptlText.y = rectlClient.yTop - pti -> usTopIncr;
```

```
            ulBegin = pti -> sVOffset;
            ulEnd = ulBegin + rectlClient.yTop / pti -> usVIncr;

            for (i = 0; i < ulBegin; i++)
               {
                ulLength = strlen (pWork);
                pWork += ulLength;
                pWork += 2;
               }

            for (i = ulBegin; i < ulEnd; i++)
               {
                ulLength = strlen (pWork);
                GpiCharStringAt (hps, &ptlText, ulLength, pWork);
                pWork += ulLength;
                pWork += 2;
                ptlText.y -= pti -> usVIncr;
               }
           }

        WinEndPaint (hps);
       }
       break;

    case WM_VSCROLL:

    /*********************************************************************/
    /*                                                                   */
    /* Vertical scrolling support is provided by the function fnVScroll. */
    /*                                                                   */
    /*********************************************************************/

       pti = WinQueryWindowPtr (hwnd, 0);

       pti -> sVOffset = fnVScroll (pti -> hwndVS, hwnd,
                                    pti -> usVIncr, pti -> ulHeight, mp2);
       break;

    case WM_HSCROLL:

    /*********************************************************************/
    /*                                                                   */
    /* Horiz. scrolling support is provided by the function fnHScroll.   */
    /*                                                                   */
    /*********************************************************************/

       pti = WinQueryWindowPtr (hwnd, 0);

       pti -> sHOffset = fnHScroll (pti -> hwndHS, hwnd,
                                    pti -> usHIncr, pti -> ulWidth, mp2);
       break;

    case WM_CHAR:

    /*********************************************************************/
    /*                                                                   */
    /* Keyboard messages are passed on to the horizontal or vertical     */
    /* scroll bars if necessary, by the function fnCharScroll.           */
    /*                                                                   */
    /*********************************************************************/

       pti = WinQueryWindowPtr (hwnd, 0);

       fnCharScroll (pti -> hwndVS, pti -> hwndHS, msg, mp1, mp2);
       break;

    case WM_COMMAND:

    /*********************************************************************/
    /*                                                                   */
    /* Action bar commands. Determine the command and handle it.         */
    /*                                                                   */
    /*********************************************************************/

       pti = WinQueryWindowPtr (hwnd, 0);

       switch (SHORT1FROMMP (mp1))
          {
           case IDM_OPEN:

           /*************************************************************/
```

```
/*                                                                  */
/* Open a new file. Show the file open dialog, and if the           */
/* user selects a file and hits OK, then kick off a thread to       */
/* do the loading. The thread takes as argument a pointer to        */
/* a data structure that contains the file name and the client      */
/* window handle, for message posting back.                         */
/*                                                                  */
/********************************************************************/
    {
     PFILEDLG  pfd;
     POPENINFO poi;
     TID       tidOpen;

     pfd = (PFILEDLG) malloc (sizeof (FILEDLG));
     memset (pfd, 0, sizeof (FILEDLG));
     pfd -> cbSize = sizeof(FILEDLG);
     pfd -> fl = FDS_CENTER | FDS_OPEN_DIALOG;
     pfd -> pszTitle = "PM Text Demo";
     strcpy (pfd -> szFullFile, "*.*");

     WinFileDlg (HWND_DESKTOP, hwnd, pfd);

     if (pfd -> lReturn == DID_OK)
        {
         poi = (POPENINFO) malloc (sizeof (OPENINFO));

         strcpy (poi -> szFileName,
                 pfd -> szFullFile);

         poi -> hwnd = hwnd;

         DosCreateThread (&tidOpen, (PFNTHREAD) fnOpenThread,
                          (ULONG) poi, 0, 8192);
        }
     free (pfd);
    }
    break;

 case IDM_CLEAR:

/********************************************************************/
 /*                                                                 */
 /* Get rid of the file currently being displayed, if any.          */
 /*                                                                 */
/********************************************************************/

    if (pti -> boolText)
       {
        RECTL rcl;

        pti -> boolText = FALSE;
        DosFreeMem (pti -> pFile);

        WinQueryWindowRect (hwnd, &rcl);

        pti -> ulWidth = 0;
        pti -> ulHeight = 0;

        fnFixScrollSize (pti -> hwndVS, pti -> hwndHS,
                         pti -> ulWidth, pti -> ulHeight,
                         rcl.xRight, rcl.yTop,
                         pti -> usVIncr, pti -> usHIncr,
                         &(pti -> sVOffset), &(pti -> sHOffset));
        WinInvalidateRect (hwnd, NULL, 0);
       }
    break;

 case IDM_EXIT:

/********************************************************************/
 /*                                                                 */
 /* User wants to end the application; post a WM_CLOSE to get       */
 /* the message box confirmation first.                             */
 /*                                                                 */
/********************************************************************/

    WinPostMsg (hwnd, WM_CLOSE, 0, 0);
    break;

 default:
    break;
```

```
        }
        break;

    case UM_FILEERROR:

    /*********************************************************************/
    /*                                                                   */
    /* An error occurred loading the file - just notify the user with a  */
    /* message box, and take no further action.                          */
    /*                                                                   */
    /*********************************************************************/

       WinMessageBox (HWND_DESKTOP, hwnd, "Error loading file",
                      "Text Browser", 1, MB_OK | MB_MOVEABLE | MB_WARNING);
       break;

    case UM_FILEREAD:

    /*********************************************************************/
    /*                                                                   */
    /* The file has been successfully loaded. Free the data buffer from  */
    /* the previous file, then reset the scroll bars according to the    */
    /* number of lines and maximum line width provided in mp2. The new   */
    /* file buffer is pointed to from mp1. Repaint the window with this  */
    /* file.                                                             */
    /*                                                                   */
    /*********************************************************************/
       {
        RECTL rcl;

        pti = WinQueryWindowPtr (hwnd, 0);
        WinQueryWindowRect (hwnd, &rcl);

        if (pti -> boolText)
           DosFreeMem (pti -> pFile);

        pti -> boolText = TRUE;
        pti -> pFile = PVOIDFROMMP (mp1);
        pti -> ulWidth = (LONG) SHORT1FROMMP (mp2);
        pti -> ulHeight = (LONG) SHORT2FROMMP (mp2);

        fnFixScrollSize (pti -> hwndVS, pti -> hwndHS,
                         pti -> ulWidth, pti -> ulHeight,
                         rcl.xRight, rcl.yTop,
                         pti -> usVIncr, pti -> usHIncr,
                         &(pti -> sVOffset), &(pti -> sHOffset));

        WinInvalidateRect (hwnd, 0, FALSE);
       }
       break;

    case WM_CLOSE:

    /*********************************************************************/
    /*                                                                   */
    /* Close from the system menu - prompt for confirmation, then post a */
    /* WM_QUIT message to end the application.                           */
    /*                                                                   */
    /*********************************************************************/

       if (WinMessageBox (HWND_DESKTOP, hwnd,
                          "Are you sure you want to quit?",
                          "Text Browser",
                          1, MB_YESNO | MB_ICONQUESTION | MB_MOVEABLE)
             == MBID_YES)
       WinPostMsg (hwnd, WM_QUIT, 0, 0);
       break;

    case WM_DESTROY:

    /*********************************************************************/
    /*                                                                   */
    /* Upon window destruction, free up the instance data structure and  */
    /* the file buffer if it exists.                                     */
    /*                                                                   */
    /*********************************************************************/

       pti = WinQueryWindowPtr (hwnd, 0);
       if (pti -> boolText)
          DosFreeMem (pti -> pFile);
       free (pti);
```

```
          break;

        default:

          mr = WinDefWindowProc (hwnd, msg, mp1, mp2);
          break;
       }

 return mr;
}

/*****************************************************************************/
/*                                                                           */
/* Vertical scrolling function: Scroll by 1 char if line scrolling; else     */
/* page scrolling. Set the scroll bar to the new location, and return the    */
/* new scroll offset value to the client window procedure to re-draw the     */
/* text at the newly scrolled position.                                      */
/*                                                                           */
/*****************************************************************************/

SHORT fnVScroll (HWND hwndVScroll, HWND hwndClient, USHORT usVIncr,
                 USHORT cyData, MPARAM mp2)
{
 SHORT sVOffset;
 RECTL rclClient;

 sVOffset = SHORT1FROMMR (WinSendMsg (hwndVScroll, SBM_QUERYPOS, 0, 0));
 WinQueryWindowRect (hwndClient, &rclClient);

 switch (SHORT2FROMMP (mp2))
    {
     case SB_LINEUP:
        sVOffset = max (0, (sVOffset - 1));
        break;
     case SB_LINEDOWN:
        sVOffset = min ((sVOffset + 1),
                        (cyData - rclClient.yTop / usVIncr));
        break;
     case SB_PAGEUP:
        sVOffset = max (0, (sVOffset - rclClient.yTop / usVIncr));
        break;
     case SB_PAGEDOWN:
        sVOffset = min ((sVOffset + rclClient.yTop / usVIncr),
                        (cyData - rclClient.yTop / usVIncr));
        break;
     case SB_SLIDERTRACK:
        sVOffset = SHORT1FROMMP (mp2);
        break;
     default:
        break;
    }

 if (sVOffset != SHORT1FROMMR (WinSendMsg (hwndVScroll, SBM_QUERYPOS,
                                           0, 0)))
    {
     WinSendMsg (hwndVScroll, SBM_SETPOS, MPFROMSHORT (sVOffset), 0);
     WinInvalidateRect (hwndClient, NULL, FALSE);
    }

 return (sVOffset);
}

/*****************************************************************************/
/*                                                                           */
/* Horizontal scrolling - much like vertical, only sideways.                 */
/*                                                                           */
/*****************************************************************************/

SHORT fnHScroll (HWND hwndHScroll, HWND hwndClient, USHORT usHIncr,
                 USHORT cxData, MPARAM mp2)
{
 SHORT sHOffset;
 RECTL rclClient;

 sHOffset = (USHORT) WinSendMsg (hwndHScroll, SBM_QUERYPOS, 0, 0);
 WinQueryWindowRect (hwndClient, &rclClient);

 switch (SHORT2FROMMP (mp2))
    {
     case SB_LINELEFT:
        sHOffset = max (0, (sHOffset - 1));
```

```
            break;
         case SB_LINERIGHT:
            sHOffset = min ((sHOffset + 1),
                            (cxData - rclClient.xRight / usHIncr));
            break;
         case SB_PAGELEFT:
            sHOffset = max (0, (sHOffset - rclClient.xRight / usHIncr));
            break;
         case SB_PAGERIGHT:
            sHOffset = min ((sHOffset + rclClient.xRight / usHIncr),
                            (cxData - rclClient.xRight / usHIncr));
            break;
         case SB_SLIDERTRACK:
            sHOffset = SHORT1FROMMP (mp2);
            break;
         default:
            break;
        }

    if (sHOffset != SHORT1FROMMR (WinSendMsg (hwndHScroll, SBM_QUERYPOS,
                                              0, 0)))
        {
         WinSendMsg (hwndHScroll, SBM_SETPOS, MPFROMSHORT (sHOffset), 0);
         WinInvalidateRect (hwndClient, NULL, FALSE);
        }

    return (sHOffset);
}

/**************************************************************************/
/*                                                                        */
/* Character scrolling: only if the keystrokes are arrow or page keys,    */
/* pass the messages to the scroll bar to handle automatically.           */
/*                                                                        */
/**************************************************************************/

VOID fnCharScroll (HWND hwndVScroll, HWND hwndHScroll,
                   USHORT msg, MPARAM mp1, MPARAM mp2)
{
 switch (SHORT2FROMMP (mp2))
    {
     case VK_LEFT:
     case VK_RIGHT:
        WinSendMsg (hwndHScroll, msg, mp1, mp2);
        break;

     case VK_UP:
     case VK_DOWN:
     case VK_PAGEUP:
     case VK_PAGEDOWN:
        WinSendMsg (hwndVScroll, msg, mp1, mp2);
        break;
     default:
        break;
    }
}

/**************************************************************************/
/*                                                                        */
/* Fix scroll bar proportions if the window size or font changes: send the */
/* scroll bars a message with the new proportion to figure out.           */
/*                                                                        */
/**************************************************************************/

VOID fnFixScrollSize (HWND hwndVScroll, HWND hwndHScroll,
                      USHORT cxData, USHORT cyData,
                      USHORT cxWindow, USHORT cyWindow,
                      USHORT usVIncr, USHORT usHIncr,
                      PSHORT psVOffset, PSHORT psHOffset)
{
 USHORT        usHeight;
 USHORT        usWidth;

 usHeight = cyWindow / usVIncr;
 usWidth = cxWindow / usHIncr;

 if (usHeight >= (cyData - *psVOffset) && *psVOffset != 0)
    *psVOffset = max (0, (cyData - usHeight));

 if (usWidth >= (cxData - *psHOffset) && *psHOffset != 0)
    *psHOffset = max (0, (cxData - usWidth));
```

```
    WinSendMsg (hwndVScroll, SBM_SETTHUMBSIZE,
                MPFROM2SHORT (usHeight, cyData), 0);

    WinSendMsg (hwndHScroll, SBM_SETTHUMBSIZE,
                MPFROM2SHORT (usWidth, cxData), 0);

    WinSendMsg (hwndVScroll, SBM_SETSCROLLBAR,
                MPFROMSHORT (*psVOffset),
                MPFROM2SHORT (0, (cyData - usHeight)));

    WinSendMsg (hwndHScroll, SBM_SETSCROLLBAR,
                MPFROMSHORT (*psHOffset),
                MPFROM2SHORT (0, (cxData - usWidth)));
}

/****************************************************************************/
/*                                                                          */
/* FIle open thread function - Open the file, allocate memory, read it in, */
/* parse to zero-terminated strings, and post a message back to the client */
/* window.                                                                  */
/*                                                                          */
/****************************************************************************/

VOID fnOpenThread (POPENINFO pOpenInfo)
{
 HFILE              hFile;
 PCH                pFile;
 ULONG              ulAction;
 FILESTATUS         fstsFile;
 ULONG              ulBytes;
 CHAR               szFileName[255];
 HWND               hwnd;
 BOOL               boolError;
 PCH                pWork;
 USHORT             i, usWidth, usHeight;

 strcpy (szFileName, pOpenInfo -> szFileName);
 hwnd = pOpenInfo -> hwnd;
 free (pOpenInfo);
 boolError = FALSE;

 do {
    /*********************************************************************/
    /*                                                                   */
    /* Open the file, privately to this process. Open for read only, and */
    /* fail if the file does not exist.                                  */
    /*                                                                   */
    /*********************************************************************/
    if (DosOpen (szFileName, &hFile, &ulAction, 0, 0,
                 OPEN_ACTION_FAIL_IF_NEW |
                 OPEN_ACTION_OPEN_IF_EXISTS,
                 OPEN_SHARE_DENYWRITE |
                 OPEN_ACCESS_READONLY,
                 0))
       {
        boolError = TRUE;
        break;
       }

    /*********************************************************************/
    /*                                                                   */
    /* Determine the size of the file, from a FILESTATUS data structure. */
    /*                                                                   */
    /*********************************************************************/

    if (DosQueryFileInfo (hFile, FIL_STANDARD, &fstsFile,
                          sizeof (FILESTATUS)))
       {
        boolError = TRUE;
        break;
       }

    /*********************************************************************/
    /*                                                                   */
    /* Allocate sufficient memory to hold the contents of the file.      */
    /*                                                                   */
    /*********************************************************************/

    if (DosAllocMem ((PPVOID) &pFile, fstsFile.cbFileAlloc,
                     PAG_WRITE | PAG_COMMIT))
```

```
        {
         boolError = TRUE;
         break;
        }

     /*********************************************************************/
     /*                                                                   */
     /* Read the file into the memory buffer just allocated.              */
     /*                                                                   */
     /*********************************************************************/

     if (DosRead (hFile, (PVOID) pFile, fstsFile.cbFile, &ulBytes))
        {
         boolError = TRUE;
         DosFreeMem (pFile);
         break;
        }

     /*********************************************************************/
     /*                                                                   */
     /* The file could be handled any way you want - for ease of drawing  */
     /* the text, just change the carriage returns to 0's so that the file */
     /* can be looked at as a sequence of zero-terminated strings. Also,  */
     /* this lets me calculate the height and width of the file, namely,  */
     /* the number of lines and the maximum line width, in order to set   */
     /* the scroll bars correctly.                                        */
     /*                                                                   */
     /*********************************************************************/

     pWork = pFile;

     usWidth = usHeight = i = 0;

     while (pWork < pFile + fstsFile.cbFile)
        {
         if (*pWork == '\r')
            {
             *pWork = '\0';
             pWork++;
             usHeight++;
             if (i > usWidth)
                usWidth = i;
             i = 0;
            }
         pWork++;
         i++;
        }

    } while (FALSE);

 /*************************************************************************/
 /*                                                                       */
 /* Close the file when finished. If an error occurred at any step, post  */
 /* the user defined message UM_FILEERROR to the main window procedure to */
 /* indicate failure. Otherwise, post the user defined message UM_FILEREAD */
 /* to indicate success and allow the main window to display the file.    */
 /*                                                                       */
 /*************************************************************************/

 DosClose (hFile);

 if (boolError)
    WinPostMsg (hwnd, UM_FILEERROR, 0, 0);
 else
    WinPostMsg (hwnd, UM_FILEREAD, MPFROMP (pFile),
                MPFROM2SHORT (usWidth, usHeight));

 DosExit (EXIT_THREAD, 0);
}
```

The following is the MAKE file for browse2:

```
BROWSE2.MAK

browse2.exe: browse2.obj browse2.def browse2.res
        link386 /NOI /DEBUG browse2.obj,,,,browse2.def;
        rc browse2.res browse2.exe

browse2.res: browse2.rc browse2.h
        rc -r browse2.rc
```

```
browse2.obj: browse2.c browse2.h
        icc /c /Ss /Ti+ /Kb /Gm browse2.c
```

The following is the resource file for browse2:

BROWSE2.RC

```
#include <os2.h>
#include "browse2.h"

MENU ID_FRAME
   {
    SUBMENU "~File", IDM_FILE
       {
        MENUITEM "~Open...",      IDM_OPEN
        MENUITEM "~Clear",        IDM_CLEAR
        MENUITEM SEPARATOR
        MENUITEM "E~xit...",      IDM_EXIT
       }
   }
```

The following is the module definition file for browse2:

BROWSE2.DEF

```
;------------------------------------
; BROWSE2.DEF module definition file
;------------------------------------

NAME            BROWSE2  WINDOWAPI

DESCRIPTION     'Text File browser - FAK 08/93'
STACKSIZE       8192
```

The following is browse2.h, the header file for browse2:

BROWSE2.H

```
/*****************************************************************************/
/*                                                                           */
/* Text browser demo - application header.                                   */
/*                                                                           */
/*****************************************************************************/

#define CLIENT_CLASS_NAME  "WC_CLIENT" // Registered client class name

#define ID_FRAME        100            // Frame window (resource) ID

#define IDM_FILE        200            // Menu IDs
#define IDM_OPEN        201
#define IDM_CLEAR       202
#define IDM_EXIT        203

typedef struct                  // Client Window instance data structure
   {
    HWND   hwndVS;              // Vertical scroll bar window handle
    HWND   hwndHS;              // Horizontal scroll bar window handle
    USHORT usVIncr;             // Vertical spacing (line height)
    USHORT usHIncr;             // Horizontal spacing (max character width)
    USHORT usTopIncr;           // Spacing from top of window
    SHORT  sVOffset;            // Offset of lines above top
    SHORT  sHOffset;            // Offset of characters beside left
    ULONG  ulHeight;            // Total number of lines
    ULONG  ulWidth;             // Maximum characters per line
    BOOL   boolText;            // Is text being drawn?
    PVOID  pFile;               // Pointer to text
   } TEXTINFO, *PTEXTINFO;

typedef struct                  // Thread argument data structure
   {
    HWND  hwnd;                 // Client window handle
```

```
    CHAR  szFileName [255];  // File name
   } OPENINFO, *POPENINFO;

// User-defined messages

#define UM_FILEREAD  WM_USER + 1 // File successfully loaded
#define UM_FILEERROR WM_USER + 2 // File load error occurred

// Function prototypes

MRESULT wpClientWndProc(HWND hwnd, ULONG msg, MPARAM mp1, MPARAM mp2);

#pragma linkage (wpClientWndProc, system)

SHORT fnVScroll (HWND hwndVScroll, HWND hwndClient, USHORT usVIncr,
                 USHORT cyData, MPARAM mp2);

SHORT fnHScroll (HWND hwndHScroll, HWND hwndClient, USHORT usHIncr,
                 USHORT cxData, MPARAM mp2);

VOID fnCharScroll (HWND hwndVScroll, HWND hwndHScroll,
                   USHORT msg, MPARAM mp1, MPARAM mp2);

VOID fnFixScrollSize (HWND hwndVScroll, HWND hwndHScroll,
                      USHORT cxData, USHORT cyData,
                      USHORT cxWindow, USHORT cyWindow,
                      USHORT usVIncr, USHORT usHIncr,
                      PSHORT psVOffset, PSHORT psHOffset);

VOID fnOpenThread (POPENINFO pOpenInfo);

#pragma linkage (fnOpenThread, system)
```

Index

Symbols

Numerics

A

B

C

D

E

F

G

H

P

Q

R

S

T

U

V

W

Z

Introduction to OS/2 – Programming

There are two revolutions sweeping the country today. The 90s will be known as the start of the digital age. Digital technology is no longer limited to computing. It is now the basis for music in the form of CD players for our homes and cars, video in the new high definition television, and photography in the form of filmless cameras.

At the same time, the face of computing is also changing with the increasing popularity of the graphical user interface or GUI. The windows concept, first made popular on the Apple Lisa computer, is now available for all systems.

OS/2 Version 2, is the most advanced operating environment available today for integrating these technologies. A truly revolutionary system, OS/2 Version 2 is the first system designed for the new digital world.

The purpose of the OS/2 Awareness series is to help you understand how OS/2 enables you to participate in the digital and GUI revolutions. This book is a collaboration between IBM Corporation's Southwestern Area staff and One Up Corporation's staff. This volume, the second in the series, provides an introduction to OS/2 programming. It shows you how to create programs that take advantage of the powerful multitasking capabilities built into OS/2. It also shows you how to use the Presentation Manager to give your program a graphical user interface.

Volume 1 in this series provides an overview of the OS/2 system itself. It describes how you can move from the older, limited function PC-DOS and Windows environment to a fully capable OS/2 environment. Other volumes in this series tell you how to enable OS/2 to communicate effectively with remote systems, how to integrate database capabilities into OS/2, and so on.

About the Authors

Craig Chambers is a graduate of Purdue University with BS and MS degrees. He worked for IBM for 23 years in various positions including; large account System Engineer, 3270 and 3270-PC technical support, and PC-DOS and OS/2 technical support. During this time, he wrote several articles for various technical journals, made presentations at COMMON, and participated in several IBM Field Television Network broadcasts.

Feite Kraay spent three years at IBM Canada Education developing and teaching courses on OS/2 and Presentation Manager programming, from version 1.1 through 2.0. In early 1992 he joined One Up Corporation, continuing his OS/2 education and consulting activities. He is now the general manager of One Up Computer Services Ltd., the Canadian subsidiary of One Up Corporation. Feite Kraay holds a Bachelor of Mathematics degree from University of Waterloo.

Larry Pollis spent 10 years with Rolm. From there, he joined IBM, working i OS/2 technical support group. He wrote several technical bulletins and pres several topics on the IBM Field Television Network about OS/2.

Printed In U.S.A.

Price

S246-0106-00

Printed in U.S.A.