Introduction to OS/2 Warp Programming

Student Guide

Course Code OS290

by

Les Bell and Associates Pty Ltd

Version 2.9

Date 12 January, 2012

License

Authors

- The original author of this course was Les Bell and Associates Pty Ltd on 1997.
- The content was released by Les Bell and Associates Pty Ltd. under CC license on January of 2012.
- Martín Itúrbide from OS2World.com transformed the content to a newer format from Lotus Freelance and Word for OS/2 on January 2012. Martín Itúrbide thanks the people at #netlabs and Neil Waldhauer that helped to convert the older files.

Les Bell

Les Bell is widely regarded as Australia's leading authority on microcomputers. Following studies in Cybernetics and Instrument Physics at Reading University in the UK, he has followed a diverse career within the computer industry. His initial assignment to write about microcomputers for Electronics Today International magazine in London led to further editorial postings in Canada and Australia, where he was founding editor of Your Computer, Australia's leading microcomputer magazine.

In 1981 he founded Les Bell and Associates, a consultancy specialising in software development and microcomputer evaluation, selection and implementation. Clients include the Federal and State Governments and major businesses.

As a consulting editor of Your Computer, Les was convenor of the panel which selects the PC of the Year. He has written several tutorial series on BASIC, dBASE, the C programming language and other topics. He has also taught Computing and Management to business students.

Les has presented papers to the Institute of Instrumentation and Control Australia, the Australian Computer Society, SHARE/Guide and other professional bodies. He is in great demand as a public speaker and makes frequent radio and TV appearances.

As a software developer, Les has been using the OS/2 Software Development Kit since 1987, and has wide experience with other multi-user and multi-tasking operating systems. He has worked with OS/2 2.0 since 1990.

In his spare time, Les is an enthusiastic private pilot with Multi-Engine Command Instrument Rating and has even logged one hour of air combat time!

Les Bell and Associates Pty Ltd
PO Box 297
Neutral Bay Junction NSW 2089

Tel: (02) 9451 1144
Fax: (02) 9451 1122
Compuserve 71210,104
Internet: lesbell@lesbell.com.au
Home page: http://www.lesbell.com.au

Table of Contents

Introduction

This course is intended to provide programmers with a comprehensive overview of the features and concepts of OS/2 2.1. The course utilises lecture sessions interspersed with laboratory exercises to cover essential features of both OS/2 Presentation Manager and the underlying base operating system, or Control Program.

These notes are intended for reference after the course, or for reference use during lab sessions to reinforce topics introduced during the lectures. The slides used during the lectures can be found at the rear of the manual, printed four slides to each page, providing space for your notes and comments.

The course is designed to teach concepts of the OS/2 Control Program and Presentation Manager, together with specific API's and design techniques, using the C programming language. The API's documented can be called from C++ code, and the design techniques documented here can be used with C++ programs, so that the resultant application may use object-oriented techniques and classes for internal business logic and conventional C calls to the PM and OS/2 API's.

This course does *not* cover programming in C++, using the Visual Age C++ Open Class Library and/or Visual Builder for user interface implementation. However, after completing this course you will be in a much better position to utilise these tools than the novice with no PM programming experience. In addition, experience shows that while Open Class can simplify user interface coding, it does so at considerable cost in run-time performance and some loss of flexibility.

The reader might be surprised that around one-third of the course is spent on the OS/2 base operating system. There are two reasons for this:

a) It is the features of the base operating system, such as preemptive multitasking and multithreading, interprocess protection and interprocess communication, that distinguish OS/2 applications from Windows applications and

b) Well-written Presentation Manager programs rely heavily on some of the base operating system features such as threads.

The Development Process

Most programs for OS/2 are written in C, with some development now being undertaken in C++. The rest is done using high-level tools such as GPF, AME and others, which either generate executables or C code.

The simplest type of OS/2 program is a simple, single-module C program which uses C function library calls for I/O and makes no direct use of OS/2 API's. Such a program can be compiled and linked for OS/2 in much the same way as for DOS.

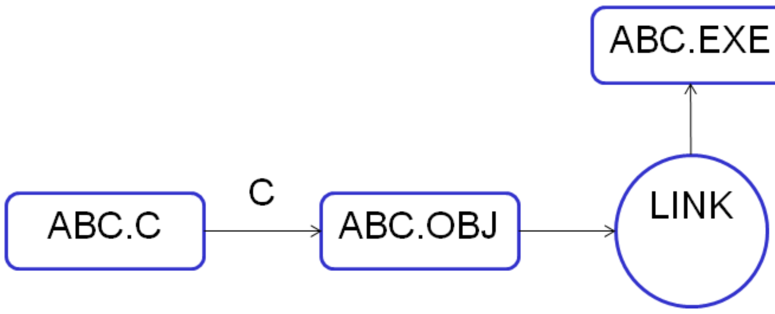


Figure 1: Simple compilation of a single source code module.

A simple program, which uses C run-time library functions for I/O, can be written as a single source module, then compiled and linked. The compile command depends upon the compiler; for 16-bit OS/2 one might use the MS C 6.00 compiler, and the command would be

```
CL /c abc.c
```

while for the IBM Visual Age C++ compiler, the command would be

```
icc /c abc.c
```

In both cases, the /c option means 'compile only, do not link' - without this, the compiler would automatically invoke the linker. In these diagrams, the C run-time library has not been shown, for compactness.

More complex applications, however, typically benefit from having the source code separated into multiple source modules, both to take advantage of the scoping rules of the language and also to allow for reuse. In addition, some low-level modules may have to be written in assembler.

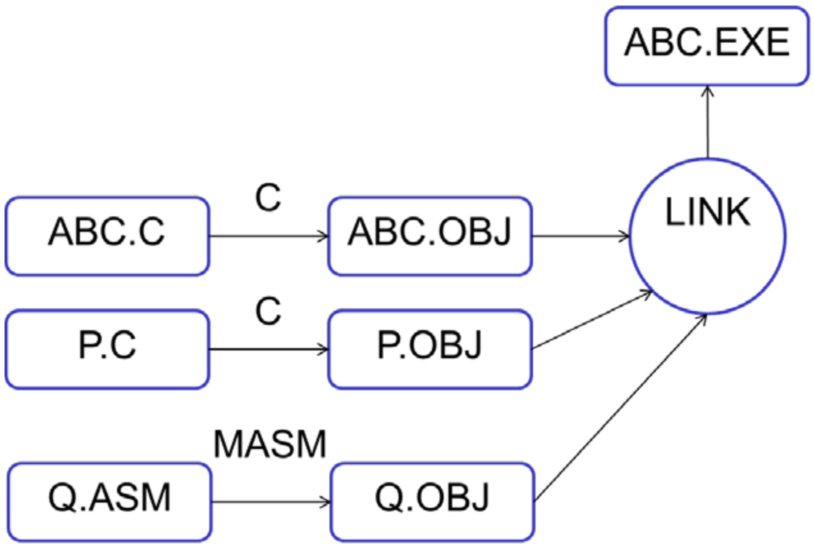


Figure 2: Compiling and linking separate modules.

For OS/2, the choice of assemblers is somewhat restricted. Most developers use Microsoft's MASM 6.0 - the difficulty here is that this is an old version and, as of MASM 6.01, Microsoft helpfully removed OS/2 support. Alternatively, Borland's C++ compiler for OS/2 comes with that company's TASM (Turbo Assembler), though buying an entire compiler just to obtain the compiler is a lousy deal. Finally, IBM is currently beta testing a new assembler called ALP (Assembly Language Processor) which can be found on the Developer Connection CD-ROM's.

Generating and Using Dynamic Link Libraries

So far, nothing is different from running a compiler under DOS, or much different from UNIX. The major difference comes when using Dynamic Link Libraries. These will be dealt with in a later chapter of the notes, so only a very abbreviated description will be given here.

A Dynamic Link Library (DLL) is essentially a library of subroutine code which can be loaded and linked to the program as it loads or even as it runs, as opposed to the normal practice of statically linking all subroutines at compile/link time. In the example shown here, two functions, R and S, are to be compiled and linked into a DLL called XYZ.DLL.

The first problem that arises is that when R.OBJ and S.OBJ are linked, the linker assumes that the target is to be a .EXE file, and consequently complains about the absence of a `main()` function and stack segment. In addition, rather than having a single entry point (`_main()`), this DLL will have two exported entry points, for the functions R and S. These problems are dealt with by writing a text file which contains linker options, called a Module Definition File (.DEF) file.

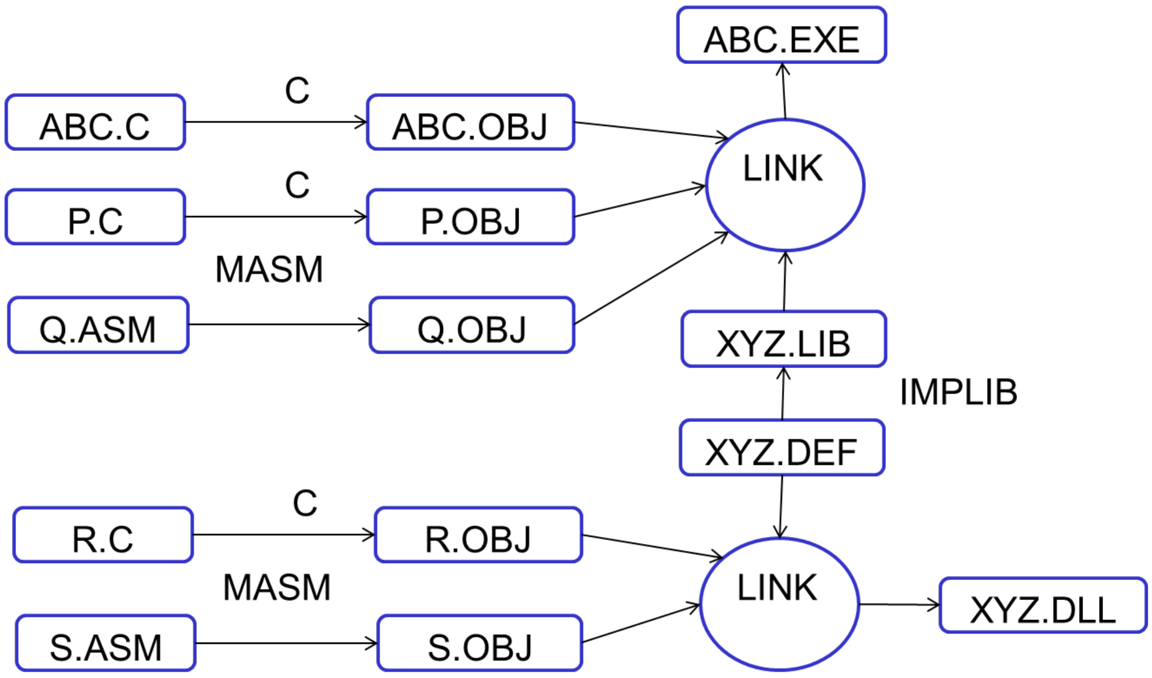


Figure 3: Compiling and linking both a Dynamic Link Library and a client program.

Now we can link the .DLL file, but when we attempt to link the main .EXE file, the linker will complain that it cannot link in the functions R and S (the actual error message will be "Unresolved external reference"). We can deal with that by creating a module definition file for ABC.EXE, or better still, by using a tool called the Import Librarian to create a .LIB file, XYZ.LIB, which contains dummy references to the functions R and S. Notice, XYZ.LIB does not contain the *code* for R and S - that would be static linking - all it contains are dummy references which cause the linker to stop complaining and generate appropriate link records in ABC.EXE which cause the system loader to also load and link XYZ.DLL.

The completed application now comprises two files: ABC.EXE and XYZ.DLL, and the developer must be careful to ship both and to provide an install program which will create the appropriate directory structure, copy the files into the right places and optionally edit the LIBPATH statement in CONFIG.SYS.

Incidentally, all OS/2 applications actually are linked in this fashion. This is because the operating system API's are accessed through dynamic linking, and every OS/2 program must be linked with the operating system's import library, which is called OS2386.LIB (16-bit apps will link with OS2286.LIB). This process is normally transparent to the programmer, however.

Presentation Manager Applications

Presentation Manager programs involve yet another level of complexity. In essence, one can view PM programs as consisting of elements such as menus, entryfields, pushbuttons, icons, scrollbars and the like, which are essentially static, together with linking code, written in C or C++, which provides the behaviour of the application in response to menu choices, pushbutton clicks, and so on. It is possible to create these interface elements through procedural calls to API's like `WinCreateWindow()`, and this is frequently done, but it is also possible to create them by specifying them in a simple language and running this through a tool called the Resource Compiler.

Essentially, one could consider the resources to be the nouns of a PM program, while the procedural code which implements behaviour contains the verbs.

This separation also allows us to put locale-specific elements such as strings, pushbutton text, menus and the like in separate files from the C source, allowing the creation of multiple versions for national language support without any modification of the code - indeed, without recompilation.

The Resource Compiler, rc.exe, is actually run at least twice: once to convert the resource compiler script file, in this case abc.rc, into a tokenized binary form, and then a second time with the -r option to bind the binary resources onto the end of the application .EXE file.

Not shown here for simplicity's sake, but quite commonly done, is the binding of resources onto dynamic link libraries. This allows, for example, common dialogs to be compiled, linked and bound with their resources and then shared by multiple applications. It also allows different sets of resources for different languages to be shipped and installed from a single media set. There are many uses for DLL's containing resources - a final example is font files, which are actually DLL's with no code or data, just the font resources.

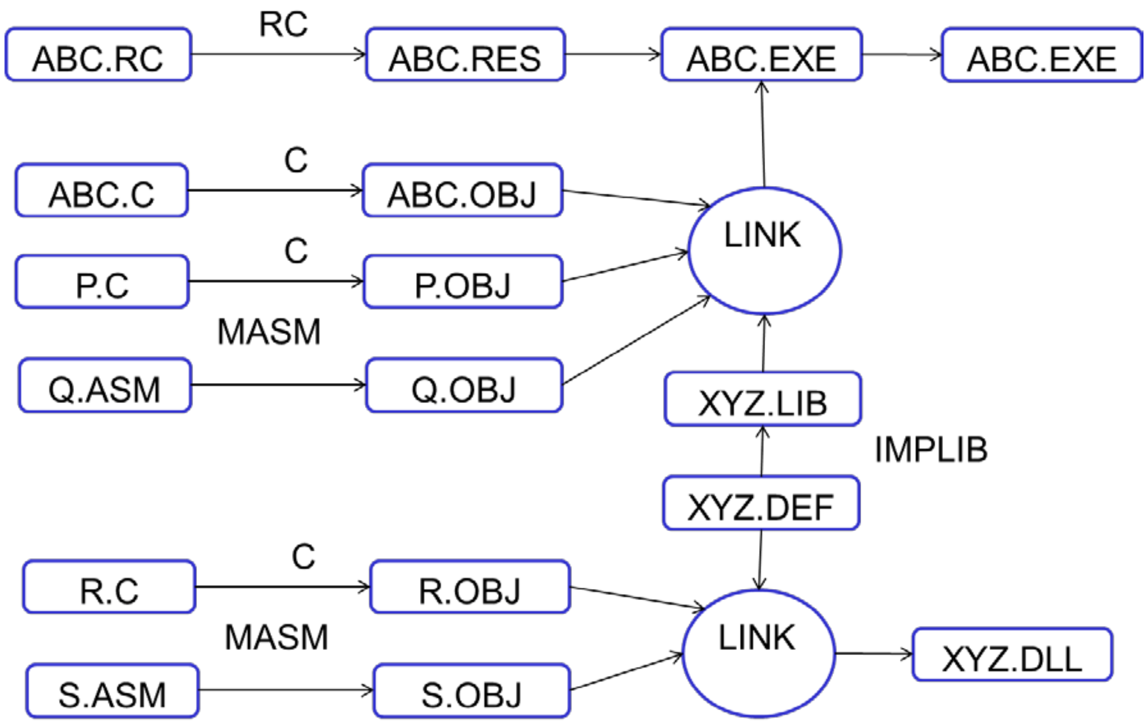


Figure 4: Compiling and linking a Presentation Manager application with resources.

Prototyping and Rapid Development

It is probably worth making clear from the outset that most developers really should not be painstakingly writing applications from scratch in C. Since programmer productivity, in lines of code per day, is roughly independent of language, and other languages can do much more, C is some way from being the ideal language when development time and cost are of prime important.

However, regardless of the language and code generation tools used, there are times when an understanding of the underlying operating system API's and principles of operation are important. This course aims to meet this need. In addition, when an application is going to be delivered to a large number of users, and robustness, stability and performance rank higher than low development cost, then C is the language of choice. It deals directly with the API's, and there is no performance overhead associated with class libraries or large runtimes. In addition, any bugs are either in your code or the operating system, whereas complex apps written using class libraries or code generators often fall foul of bugs in that third-party code.

However, for prototyping there are a number of useful tools on the marketplace:

VisPro/REXX and VX:REXX

The REXX language is a standard feature of OS/2, and is probably the most generally-useful scripting and prototyping language. All OS/2 developers should have more than passing familiarity with the REXX language, if not for application development, then because it is often encountered as the macro language in programming tools, such as the EPM editor, Visual Age C++'s Project Smarts and others. Plus, it is very useful for writing simple installation programs.

REXX's strengths:

- Interpreted tokenized code eliminates the compile/link steps for rapid testing and modification
- Essentially typeless - all variables are strings
- Excellent string handling - ideal for editing CONFIG.SYS and similar files
- REXXUTIL extensions allow access to OS/2 and Workplace Shell API's - can create objects (great for installs)
- Can be implemented as a macro language in third-party applications

The major limitation of OS/2 standard REXX is the lack of a graphical interface, which is essential for prototyping PM applications. This lack is addressed by Hockware's VisPro/REXX and Watcom's VX:REXX, both of which add support for a full range of graphical elements and an event-driven programming style. Of the two, I prefer VisPro's support for drag-and-drop programming, together with its tools for accessing DB2/2 and other databases.

The flexibility of the REXX language makes coding the logic behind VisPro/REXX forms extremely easy. For some applications, the resultant application is sufficiently powerful, fast and reliable that it can be delivered and the invoice sent - the ideal situation for the developer. If not, there are several options.

First, using techniques covered in this course, the developer with both REXX and C skills can implement his own extensions to REXX to circumvent any restrictions in the language or the visual programming tools.

Secondly, the REXX application can be used as a specification for a correct implementation in the C language.

Thirdly, if using VisPro/REXX, the interface design can be migrated to either of the VisPro/C or VisPro/C++ products, and the logic behind the interface reimplemented in the C or C++ languages. This approach is particularly attractive.

An option which will be available from late 1996 is the use of Object REXX. This updated implementation of the language allows the implementation of classes in REXX, but even more interestingly, allows OREXX programs to use SOM classes. In addition, Object REXX is the Open Scripting Architecture language for OpenDoc implemented on OS/2, which offers other interesting opportunities for prototyping methodologies.

Visual Age C++ Visual Builder

The Visual Age C++ compiler includes a Visual Builder tool which allows the visual construction of applications using the Open Class libraries. As previously mentioned, performance is nowhere near as responsive as pure C code, but for in-house applications which will have relatively few users, it is probably cheaper to provide high-performance hardware for those users than it is to employ developers to write lots of low-level optimised C code.

The Visual Builder is considerably larger and slower than the REXX visual programming tools described above (and VisPro/C/C++). The Visual Builder works by allowing the programmer to drag and drop 'parts' from a palette - some parts are visual and form part of the user interface, while others are non-visual and represent, for example, collection classes. Unlike the REXX visual programmers, Visual Builder allows the programmer to implement his own parts in C++, and IBM is encouraging the development of an after-market in parts for use with VAC++ (this parallels the existing support for Visual Age for Smalltalk).

As mentioned above, while it is possible to write simple to medium-complexity applications using just the parts provided, sooner or later additional parts will have to be written and at that point, an understanding of the underlying Presentation Manager and Control Program API's is vital in order to achieve the correct results with minimal effort.

Introduction to Tools

WorkFrame/2 1.1

The WorkFrame is the integrated development environment which ties all the other tools together. For standalone development, it is possible to work entirely within the WorkFrame. However, if you will be using SQLPREP to develop for DB2/2, or developing for SOM, you will find that the WorkFrame is of less assistance, and in particular, you will have to create and maintain your own make files.

Points to note about the WorkFrame:

The WorkFrame will automatically generate all the correct options for the compiler and link command lines, using settings notebooks for Compile Options and Link Options. However, these options only take effect when they are copied into the make file using the Tools / Make File Creation menu selection. So, when changing your setup, *remember to regenerate the make file.*

WorkFrame 3.0

WorkFrame 2.1 was a slightly problematic product which was not used in this course. It lacked, for example, support for projects with components in multiple directories and appropriate support for subprojects. However, with the release of Visual Age C++, the WorkFrame was significantly enhanced and its object-oriented design cleaned up.

The Visual Age C++ folder contains a template for VisualAge C++ Projects. When opened up a VA C++ project looks like this:

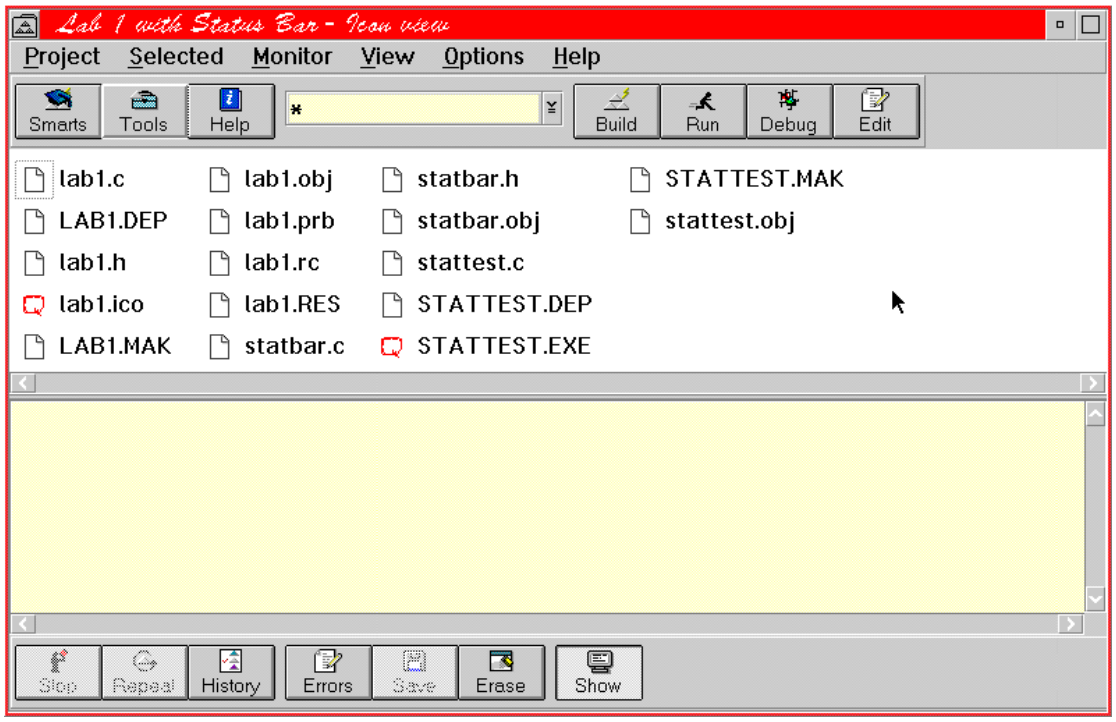
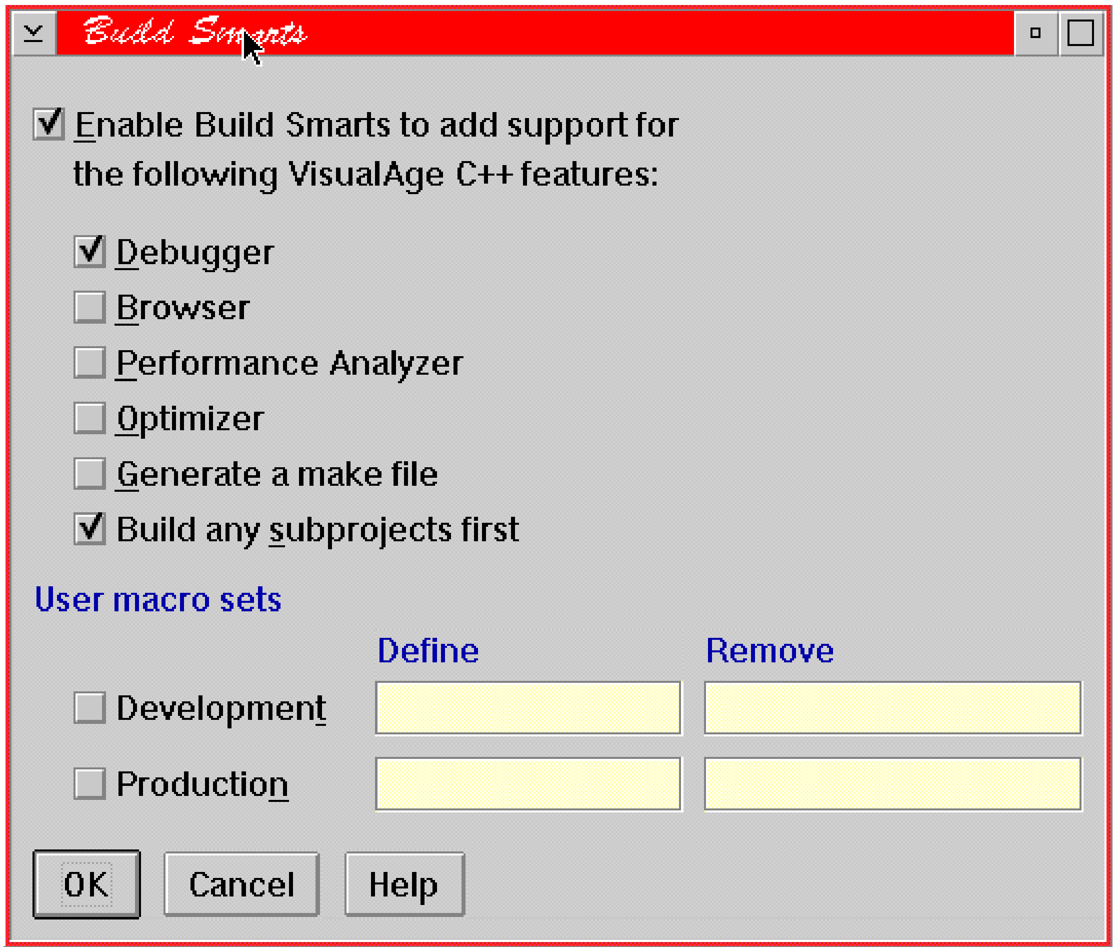


Figure 1. A VisualAge C++ Project

The top half of the main window area shows the project files, while the lower window shows the output of any operations, such as makes. The toolbar across the top of the window allows the programmer to set up 'Build Smarts', which automatically add support for project options, customise the tools such as the compiler and linker and get help. The right-hand set of buttons initiate a project build, run, debug or edit respectively.

There is an important distinction between the terms build and make in Workframe 3.0. By default, building a project will automatically cause a MakeMake, i.e. regeneration of the make files. This is a major distinction from earlier versions of the Workframe, and means that you cannot forget to rebuild the make file after changing compile and link options. However, since it happens on every build, it can slow you down significantly, and so you might want to change the 'Build Smarts' so that it does not generate a make file each time:



[✓] Enable Build Smarts to add support for the following VisualAge C++ features:

[✓] Debugger
[ ] Browser
[ ] Performance Analyzer
[ ] Optimizer
[ ] Generate a make file
[✓] Build any subprojects first

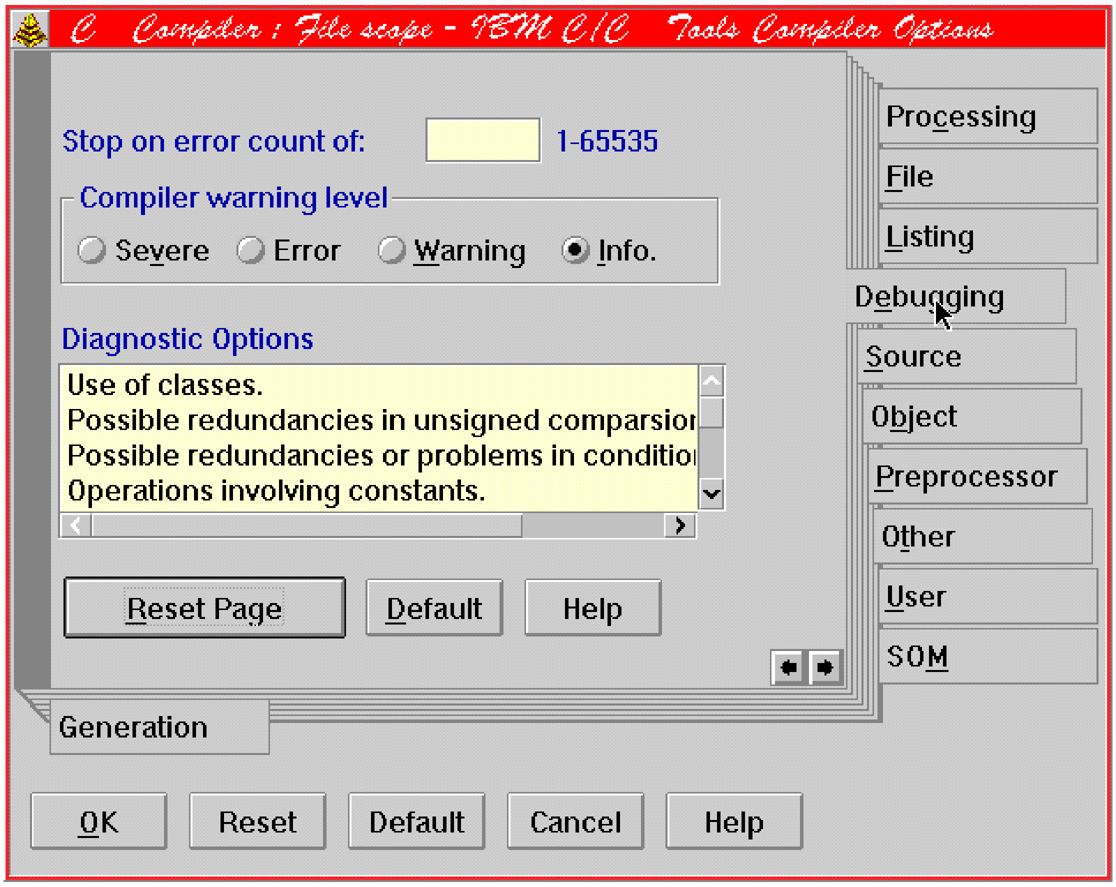
User macro sets

	Define	Remove
[ ] Development		
[ ] Production		

Figure 2. Build Smarts can add support for various Visual Age C++ tools.

Alternatively, ensure that, rather than clicking on the 'Build' button, you initiate a make operation by choosing it off the Project menu, or pressing Ctrl-Shift-M. You can also customise the project toolbar to include a 'Make' button in addition to the 'Build' one.

It is important, before attempting to compile or link a project, that you set up the compiler and linker options correctly. While this can be done by pressing the 'Tools Setup' button and then selecting the appropriate tools, a faster way is to choose Options / Compile or Options / Link off the project menu bar.



C Compiler : File scope - IBM C/C Tools Compiler Options

Stop on error count of: 1-65535

Compiler warning level

Severe Error Warning [✓ Info.]

Diagnostic Options

Use of classes.
Possible redundancies in unsigned comparsio
Possible redundancies or problems in conditio
Operations involving constants.

Figure 3. Compiler Options, showing the Debugging section

The most important Compiler options to watch for during the course are:

- On the Processing page, check that the target is a DLL if appropriate.
- Under Debugging / Generation, ensure that All is selected, so that the debugger can display source code and also variables.
- Under Source / Details, check 'Allow use of '//' for comments', as some of the lab source files have C++-style comments in them.
- Under Object, ensure that the Multithreaded library is selcted if appropriate.

We do not use any of the other options in this course, although we may discuss some of them.

The important Link Options are:

- On the Generation Page, check 'Include debug information', and set the application type as appropriate: None for a DLL, Window for a character mode application to run in a window, Full screen for a character mode application which you want to force to full screen in its own session and PM for Presentation Manager applications.
- On the File Names page, supply the names of any import libraries if the application uses dynamic link libraries (e.g. Lab 3) and, if building a DLL, provide the name of a .DEF file.

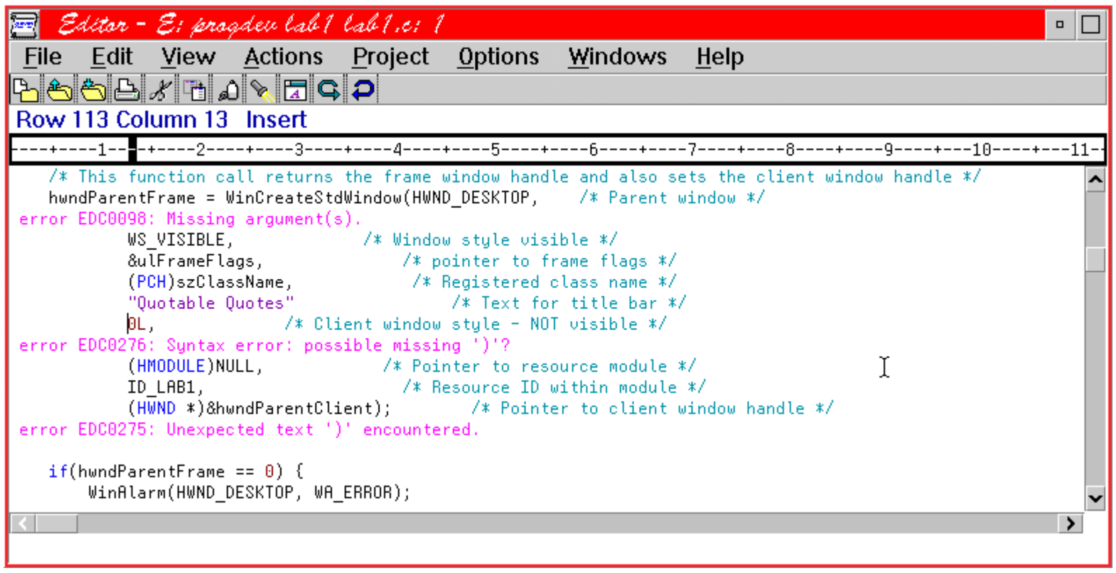
These are the only link options used in the course.

Remember, after changing options, to either do a makemake or a build to get the options reflected in the make file.

Dealing with Compile-Time Errors

If there are syntax errors in your source code, the compiler will complain and will produce error messages. Upon termination, the compiler will set the ERRORLEVEL variable to a non-zero value and this will cause the nmake utility to terminate.

Selecting a compiler output line that contains a source module name and (row:column) coordinate for the error and either double-clicking or pressing the 'Errors' button will cause the editor to be launched with the error highlighted.



Row 113 Column 13   Insert

```
----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+----8----+----9----+---10----+---11-
   /* This function call returns the frame window handle and also sets the client window handle */
   hwndParentFrame = WinCreateStdWindow(HWND_DESKTOP,    /* Parent window */
error EDC0098: Missing argument(s).
           WS_VISIBLE,             /* Window style visible */
           &ulFrameFlags,              /* pointer to frame flags */
           (PCH)szClassName,            /* Registered class name */
           "Quotable Quotes"                /* Text for title bar */
           0L,             /* Client window style - NOT visible */
error EDC0276: Syntax error: possible missing ')'?
           (HMODULE)NULL,            /* Pointer to resource module */
           ID_LAB1,                    /* Resource ID within module */
           (HWND *)&hwndParentClient);        /* Pointer to client window handle */
error EDC0275: Unexpected text ')' encountered.

   if(hwndParentFrame == 0) {
       WinAlarm(HWND_DESKTOP, WA_ERROR);
```

Figure 4. The editor (LPEX) can highlight errors and display error messages

Once the editor is open, you can use Ctrl-X to find the next error, and Alt-X to return to the previous one.

As you can see, the editor parses each line as it is typed in, and will highlight different syntactical elements of the language in different colours. In addition, it will warn of unterminated strings and other entry errors.

The editor, LPEX, also supports on-line access to the Toolkit's on-line Programming Guides and References. To obtain help for an OS/2 API or a C/C++ language statement or library function, simply place the cursor on the appropriate word and press either F1 or Ctrl-H. On-line help is also available from the editor's Help menu, including access to the extensive 'How Do I . .' files which are supplied as part of Visual Age C++.

However, you may prefer to stick with the OS/2 Enhanced Editor, EPM, for reasons of familiarity, and the WorkFrame can be customised to support this.

EPM

The EPM editor is one of a family of editors written using the E language and editor toolkit, an IBM internally-developed macro language which has been used to implement a range of editors.

Some people (like me) figure that life's too short to spend time mastering every nuance of an editor, as circumstances (like the release of a new operating system) can compel you to switch editors at any moment anyway. If that's your philosophy, then EPM will suit you, as it conforms to the conventions of mouse-driven windowed editors and can be used with virtually no training.

Others prefer to invest time in mastering an editor, secure in the knowledge that this investment will be amply repaid in improved productivity. If you feel that way, EPM will also provide hours of delight as you master its use of both REXX and E as macro languages, and its hooks to other utilities. Of course, you may also have invested heavily in another editor, in which case you will welcome EPM's ease of use.

Points to note about EPM for development:

On-line help for the OS/2 2.1 Toolkit and CSet++ are available from within EPM by placing the text cursor in the function or message name of interest and pressing Ctrl-H. If this does not work, complaining that EPMKWHLP.NDX cannot be found, then locate that file and edit CONFIG.SYS, ensuring that its subdirectory is listed in the DPATH.

Good modular programming style (and, in fact, all OS/2 and PM programming) requires that program source code be distributed through multiple source files, and the programmer needs to cross-reference between these. In EPM, ensure that you enable the ring by setting Options / Preferences / Ring Enabled, and make sure that this saved with the Options /Save options menu choice.

Opening an editor window creates a new copy of the editor (at little cost in memory, it should be noted). Each editor window can support a ring of multiple files being edited, and these can be cycled using the small ring icons at the top right of the titlebar, or using the Ctrl-P (previous) and Ctrl-N (next) keys.

In the WorkFrame, selecting multiple files and clicking on process will result in the files being loaded into the ring automatically. To add a file into the ring, press F8 and either type in the filename, select from the listbox, or use the File list button to invoke the File Open... dialog.

EPM commands can be typed into the command dialog, which is activated with Ctrl-I. To go to a particular line, invoke the command dialog and type the line number, followed by Enter.

EPM supports automatic expansion of constructs for the C language. This may be turned on by default, which can give rise to the user being unable to get out of switch statements as the editor keeps creating more cases. To turn it off, invoke the command dialog and type 'EXPAND OFF''.

After a compilation has produced some errors, double-clicking on the error message lines will invoke the editor on the relevant file and feed the error messages to it. If the editor is already running, the error line can be 'dragged' onto the editor window. Now errors can be searched for and looked up using the 'Compiler' pull-down menu.

The Visual Age C++ C and C++ Compiler

This compiler is a high-quality C and C++ compiler from IBM's Toronto labs. In this course, we shall not be using C++, for the simple reason that virtually all complex high-performance OS/2 applications and most of the samples are written in C, and the API's are designed to be called from C. After completing this course, the student should find migration to C++ to be fairly straightforward.

However, use of the Open Class Libraries, and particularly of the Visual Builder tools will result in applications which are significantly larger and slower than equivalent 'straight' PM code. Depending on the programmer's familiarity with PM, the applications will however be written faster, and this may be an over-riding consideration for corporate developers. In any case, optimal use of the class libraries will still require an understanding of the underlying PM API's and concepts.

Hints on using VA C++:

Although most applications can be developed entirely within the WorkFrame's dialogs and notebooks for compile options setup, you would be well advised to learn some of the command line options from the on-line help. This is because diagnostic messages often suggest turning on or off different command-line options, and in addition, use of SOM or SQLPREP may require hand-construction of make files. After a while, you will get to know the most common command-line options from seeing them as the compiler runs.

The compiler has some options which cause it to remain resident in memory for (typically) ten minutes after a compilation. This avoids subsequent load-time for recompiles.

LINK386

This is an enhanced version of the traditional Microsoft/IBM linker and generates flat-model 32-bit executables. Generally, its options are set up from the dialogs and notebooks in the WorkFrame, and also from a project's .DEF file.

ILINK

The ILINK linker replaces LINK386 in Visual Age C++. This new linker is much faster and has new options for generating packed code and data, as well as linking C++ executables which use templates.

IPMD

The IPMD debugger is a Presentation Manager application which supports multiple views of source code modules, with animation, breakpoints, stepping through or over functions and all the usual debugger functions. It is of general applicability, except for debugging Workplace Shell applications; as they run as part of the shell the programmer will have to invoke the debugger from within CONFIG.SYS and run PMSHELL.EXE from within the debugger. The Toolkit also includes a kernel debugger which is run over a serial port using an attached terminal.

Other Tools

The Visual Age C++ package contains a number of other utilities which are beyond the scope of this course. These include the Performance Analyzer, which can be used to instrument an application to see where it is spending most of its time to aid with optimization, the Visual Builder, which allows drag-and-drop construction of applications with almost no coding, and the Data Access Builder, which is able to generate C++ code to access DB2/2 databases. This course is intended to provide a basic framework which will allow students to investigate these tools later.

Toolkit

The Toolkit comprises the miscellaneous utilities used in OS/2 development, such as the Resource Compiler, Font Editor and Dialog Editor, together with several help files which document the API's, messages, types and structures, and the header files which implement or prototype them. It also includes a number of programming examples.

The Toolkit is supplied as part of the Visual Age C++ package; however, developers are advised that frequent updates to the Toolkit are distributed on the Developers Connection CD-ROMs, which all developers should obtain as a matter of course. Recent DevCon CD's for example, have added information and header files on DIVE, BRender and other new features.

Intel Processor Architectures

Before examining the details of the internals of OS/2, and why it works the way it does, it is necessary to understand how the Intel 8086family of processors works, as this has a fundamental bearing on why OS/2 is needed and why it does some of the things it does.

The first of the Intel 16-bit processors, the 8086, was designed with both an eye to the future and a bridge to the past. The earlier 8080/8085 family of eight-bit processors had acquired a considerable library of software, in both dedicated, or embedded, applications as well as in the emerging personal computer marketplace.

Accordingly, the design team at Intel designed the 8086 processor to be a superset of the 8080 processor. Although the 8080 is an 8-bit processor, its registers can be paired up to act as a 'pseudo' 16-bit register set.

With its sixteen-bit register pairs, and in particular a sixteen-bit stack pointer and program counter, the 8080 was able to address a maximum of 64 Kbytes of memory (because 2^{16} = 65536 = 64 Kbytes). In order to maintain compatibility, a way had to be found to allow a processor to address more memory - say, a megabyte - but still use 16-bit registers for addresses.

And so the segmented architecture was born.

One way to think of this is as a machine which implements a number of virtual 8080 processors. It splits up its megabyte of memory so that at any time, only 64 Kbytes is accessible, and then runs an 8080 program in that segment of memory. By switching to another segment of memory, we can now run another 8080 program somewhere in the same megabyte of memory, and if we can switch between segments very quickly, we now have a multi-tasking system.

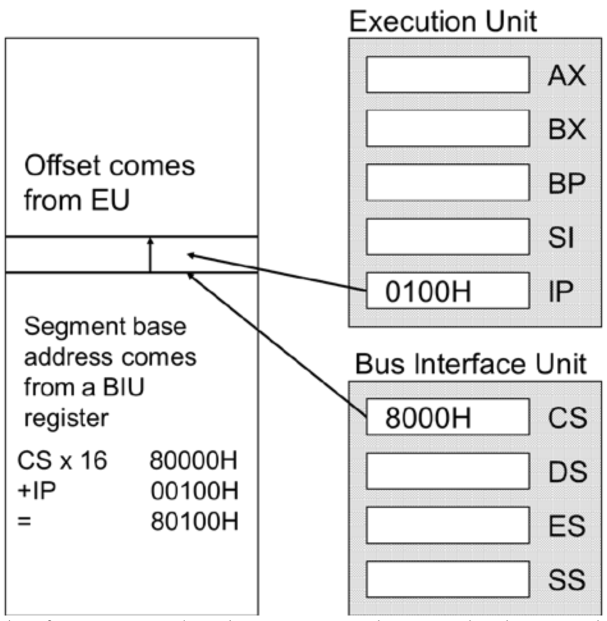


Figure 1. Simple example of a segmented architecture emulates multiple virtual 8080's.

Once this basic concept had been worked out, more refinements began to suggest themselves. For example, if we want to have two copies of the same program executing simultaneously, it makes no sense to have two complete copies of the program in memory simultaneously; one will do. But we must have two separate data areas; one for each *process*[1].

And if we have separate data areas, we should also have the ability to create separate stacks also.

Now we have the basic idea of the 8086 processor. This chip actually contains two major logic elements: the Execution Unit(EU) and the Bus Interface Unit(BIU). The execution unit is very similar to the old 8080: all the registers are primarily sixteen-bit, but can be split into their eight-bit halves for character handling, and there are more of them, together with more instructions to manipulate them, but that's all. A programmer trained on the 8080 would find the 8086 EU very familiar.

The BIU is actually a memory management unit more than being anything to do with interfacing to the external bus circuitry. It provides four registers which address the different segments of memory within which our virtual 8080 programs will execute. These registers are:

Data Segment	DS	
Extra Segment	ES	Yet another data segment
Stack Segment	SS	Used for temporary values, return addresses
Code Segment	CS	Where the instructions are stored

Of course, the master control program which is responsible for loading programs into memory and managing the multitasking must know of the existence of the segment registers, but each individual program need know nothing of their existence. A program which knows nothing about the segment registers can be relocated in memory with virtually no effort; just load it somewhere else and stick the corresponding values into the segment registers, and off it goes. (Such programs are typified by MS-DOS .COM files).

The 8086 processor is a little more complex than this, but not by much.

8086 Real Mode

The most complex aspect of the segmented architecture relates to the construction of addresses in main memory (and indeed this is the main reason for the existence of OS/2). All the processor registers are sixteen-bit in size, but as we have already seen, this allows a program to address at most 64 Kbytes of memory. How, then, can the 8086 address 1 Mbyte of memory, which will actually require 20-bit addresses to access?

Put simply, the value of a segment register is multiplied by 16 before being used as a pointer to the appropriate segment of memory. This means that each segment must start at an address which is a multiple of 16 (not a severe limitation). Such an address is called a *paragraph boundary*.

Another way of looking at this is that the BIU constructs addresses which are sent to the outside world by taking the value in a segment register, multiplying it by 16, and then adding on the value in one of the EU registers. For example, if we were about to execute a program

[1]*A process is an executing copy of a program in memory; technically, the process is not the code, but the processor registers and associated data in memory which describes the current status of this copy of the program.*

starting at location 80100H (if this doesn't make sense to you, read the section on hexadecimal numbering), the registers will contain the following values:

CS = 8000 x 16 =	80000H
IP =	0100H
	80100H

The Code Segment register addresses the base of the code segment in memory, while the IP register specifies an offset within that segment. Because all the 8086 registers are 16 bits in size, segments are restricted in size to 64 Kbytes.

This is the only mode of operation of the 8086 processor, and it is the mode in which the 80286 and 80386 processors start operation following a reset or power-up. On these processors, it is known asReal Mode .

80286 Protected Mode

However, the 80286 and 80386 processors also support other modes of operation which are more powerful and provide additional features such as interprocess memory protection and virtual memory. The simplest of these is 80286 Virtual Memory Protected Mode, usually known as just Protected Mode.

In Protected Mode, the Segment Registers of the processor do not contain the paragraph addresses of the various segments. Instead, the segment addresses are stored in a table called a segment descriptor tablealong with other segment-related information, and the segment register contents (called a segment selector) identifies a particular entry in the table.

So, to construct an address, the 80286 uses the contents of the segment register as an index into either a Global or Local Descriptor Table, and from the corresponding entry , it picks up the segment address and adds the appropriate offset in order to construct the actual address. See the diagram for an illustration of how this is done.

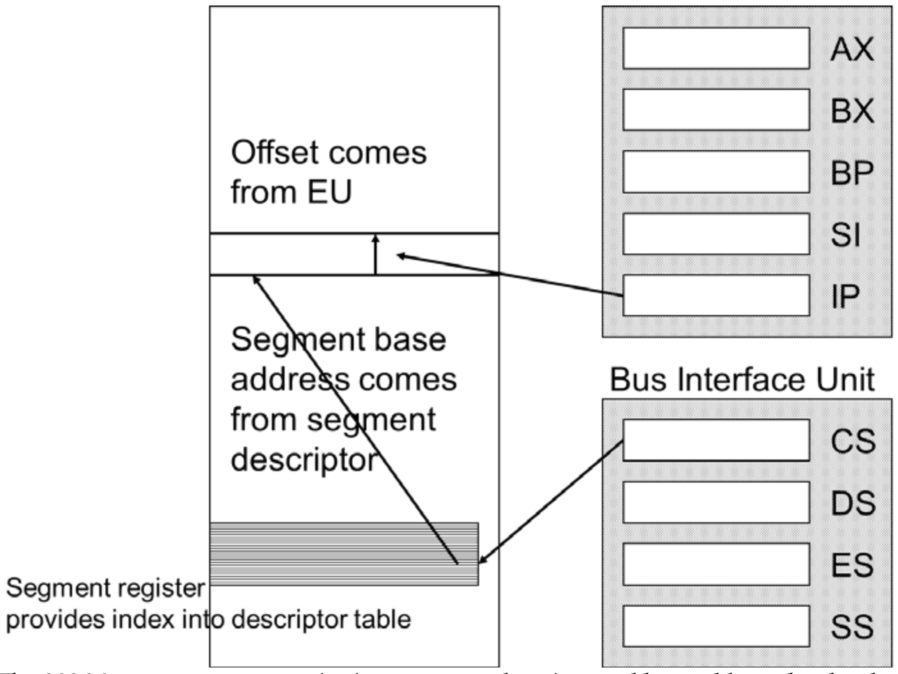

Offset comes from EU
Segment base address comes from segment descriptor
Segment register provides index into descriptor table
AX
BX
BP
SI
IP
Bus Interface Unit
CS
DS
ES
SS

Figure 2: The 80286 processor uses entries in a segment descriptor table to add another level of indirection to the addressing scheme.

Notice that, as well as the segment base address, the segment descriptor also contains some other information. The segment limitfield contains the offset of the last byte of the segment; any attempt to access an address beyond this limit causes suspension of the instruction and generation of an interrupt which vectors into the operating system. Similarly, the Access Rightsfield specifies whether the segment is read-only, execute-only and so on. Attempts to write to a read-only segment would similarly cause an exception. These features prevent a program from accessing any segments other than its own.

Incidentally, the extra address translation does not cause significantly slower operation, since the processor loads the segment entry from the LDT or GDT into an internal cache whenever a segment register is modified. The only overhead occurs when a segment register is reloaded.

Protected Mode Features Summary

The major benefits of protected mode are as follows:

Selector-based addressing: all addresses are constructed from entries in tables. There are three such table types: the Global Descriptor Table (GDT) which is the base for the operating system, one or more Local Descriptor Tables (LDT) for individual processes, and one or more Interrupt Descriptor Tables (IDT), which contain the interrupt vectors for processes.

The segment descriptors contain: the base address of each segment and the segment limit(i.e. the offset of the last byte of the segment), thus ensuring that only bytes within the segment are accessed. Also in the descriptor entry are the access rights, which contain the descriptor privilege level and segment type: Read/Write or Read Only (for data segments), and Execute/Read, or Execute Only (for code segments). A process can access data stored only at the same or a lower level segment, while a process can call services only at the same or a higher level. This is the basic principle of ring-based operating systems which assures security.

The IOPL, stored in the processor flags, similarly permits I/O instructions if it is greater than the Descriptor Privilege Level.

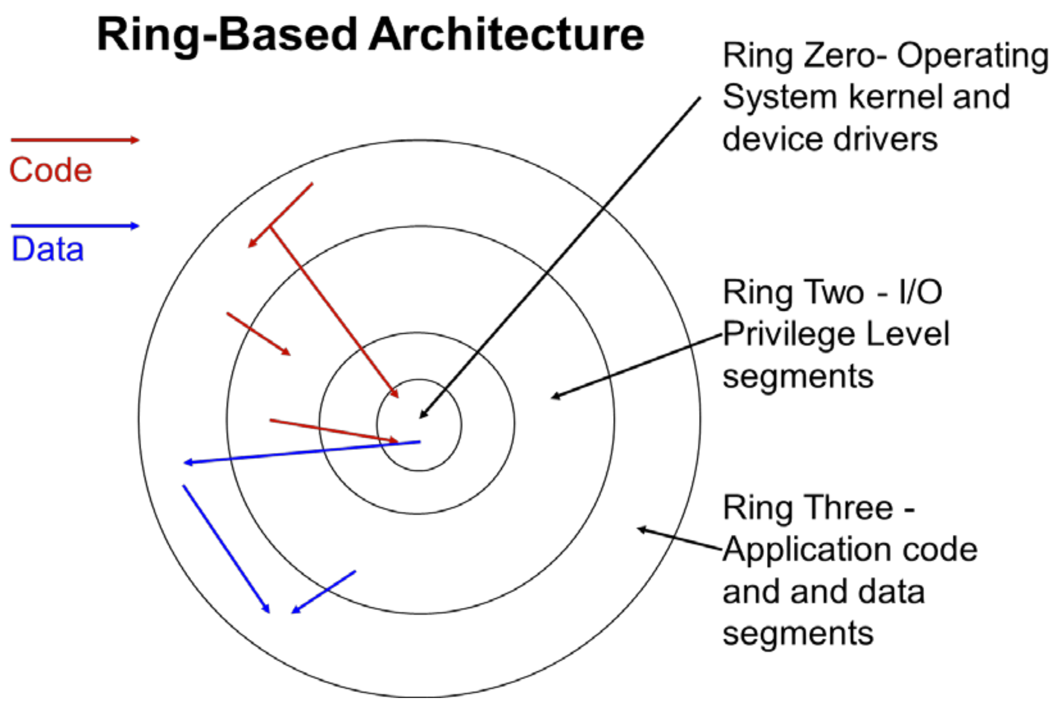

Ring-Based Architecture
Code
Data
Ring Zero- Operating System kernel and device drivers
Ring Two - I/O Privilege Level segments
Ring Three - Application code and and data segments

Figure 3: There are separate stacks for each privilege level, and the processor automatically copies parameters between the stacks.

Multi-tasking support - task state segments

Virtual memory support -

Page/segment faults

Restartable instructions

Recoverable stack faults

OS/2 and the 80386

The 80386 is a 32-bit processor which offers compatibility with the earlier 8086 and 80286 processors. The later 486, Pentium and Pentium Pro processors are based on the same architecture, and comments made here apply equally to those processors.

While OS/2 1.x runs on the 80386, it does not take advantage of many of its advanced features. In particular, when context switching, OS/2 1.x does not save and restore the upper halves of the 80386's 32-bit registers. For this reason, 32-bit code should not be run under OS/2 1.x.

The 80386 does have an instruction which will switch the processor from protected to real modes, so that it is not reliant on the kind of trickery which OS/2 perpetrates on 80286's. This means that some compatibility problems which affect 80286-based clones should not affect 80386-based machines.

OS/2 1.x does not take advantage of the 8086 Virtual Machine Mode of the 80386 in order to implement multiple DOS compatibility boxes. These features are used by OS/2 2.X, to run DOS and Windows programs.

For example, if a DOS program attempts to access video memory, the processor will look up the constructed 20-bit offset (a real-mode address) in its page translation table and discover that the page has not been allocated. This will force a trap to the operating system, which will then perform the equivalent function, probably putting the character in a window on the screen (or the equivalent in graphics mode). Likewise, attempts by terminal emulation programs to perform input/output instructions will be trapped because the virtual machine is running in ring three and does not have I/O privilege level.

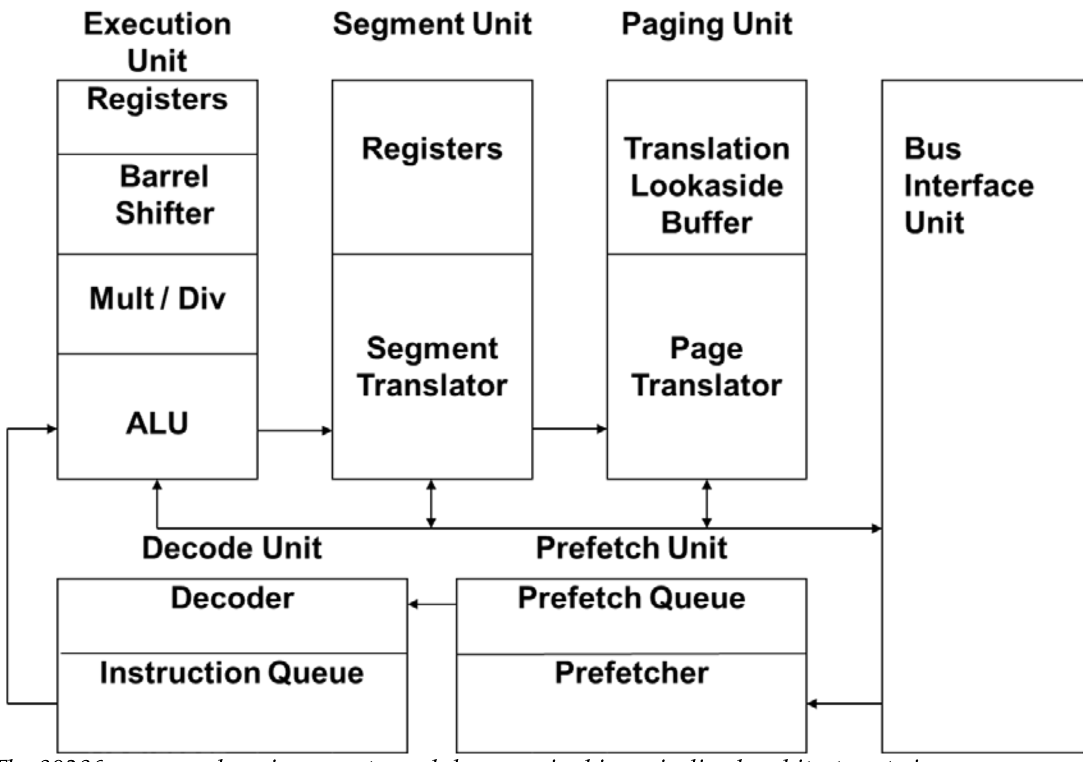
These features collectively constitute the 386's 8086 Virtual Machine Mode, which is used by OS/2 2.x and Warp to multitask DOS and Windows applications.

For OS/2 application developers, the principal benefit of the 32-bit architecture is that now the offset within a segment comes from a 32-bit register. The general-purpose registers of the 286 - AX, BX, CX, DX, SI, DI, BP, IP - are extended to 32 bits in size and renamed EAX (Extended AX), EBX, ECX . . EIP. Because the registers are 32-bit, the maximum offset is 2 ^ 32 - 1, and the maximum segment size is therefore 4 GB.

This makes it possible for all application code and data to be placed in a single segment, so that the programmer need no longer worry about the complications of memory models (small, medium, compact, large) and near/far pointers. In effect, application code is always small model - it's just that it's a bloody huge small model!

Since all code & data is in a single segment, the programmer deals only with 32-bit offsets, and ignores the segment selectors, which might as well be zero. Accordingly, this programming model is referred to as the 0:32 model, while the OS/2 1.x scheme in which far pointers consist of a 16-bit selector and 16-bit offset is referred to as 16:16.

The support for a flat 32-bit address space makes it relatively easy to port applications which were written for the UNIX operating system, and developers who are familiar with this environment can choose from a wide variety of free and shareware utilities.



The 80386 processor has six separate modules organised in a pipelined architecture to improve performance.

The 386 provides virtual memory in a different way from the 286. The segment unit generates addresses by combining the segment base address derived from a segment selector with an offset from the execution unit, in the same way as for a 286. The only difference is that while selectors are still 16-bit, offsets can be 32-bit. The address output from the segment unit is called a linear address. This is passed to the paging unit, which parses the linear address into three fields: a 10-bit page directory index, a 10-bit table index, and a 12-bit displacement.

The directory index is used to access an entry in the page directory in order to specify which page table should be used. The table index is used to specify which entry in the page table should be used. From this entry, the processor obtains a 20-bit value which is the high-order 20 bits of the resultant physical address - this is the address of the page. Finally, the 12-bit displacement is concatenated to this to produce a 32-bit physical address. You could think of the displacement as being like an offset within the 4 Kbyte page.

The system now swaps pages rather than segments. Since the pages are fixed in size (4 KB in current processors) the swap file is simply an array of 4 KB buckets, and any page can fit in any bucket, eliminating the swap file fragmentation which degrades performance on OS/2 1.x. In addition, the pages are much smaller than the swapper's buffers, so performance is further enhanced.

Programming OS/2 Presentation Manager

The OS/2 base operating system is fairly traditional in design. Although it offers many new features to the programmer who hitherto has been limited to DOS, these are mostly unremarkable. Apart from functions which have always been in DOS, though extended in some cases, the new API comprises functions for setting process priorities, memory management, I/O and the like, and are very similar to those found in systems like UNIX and VMS. The most interesting additions concern multi-threading and dynamic link libraries.

But there is one area where OS/2 is quite different from DOS, and this is the Presentation Manager graphical user interface. Corporate developers who have previously put together applications for DOS are in for a shock when they first encounter PM, as its complexity, structure and design are quite different from almost anything they will have seen before.

There is one exception: Presentation Manager is similar - at least in its window management and program structure - to Microsoft Windows. There are two differences: Windows is non-preemptive and so the programmer has to be careful to keep yielding the CPU so that other programs get a chance to run (OS/2 does this automatically) and OS/2 has many high-level graphics functions in a layer called the GPI, which Windows does not have.

Presentation Manager is a message-driven operating system within an operating system. The OS/2 base operating system, which provides the underlying services, consists of a library of functions which are called directly from application programs. PM, by contrast, is an object-oriented system in which different objects - termed windows - send each other messages, although there are still lots of functions. The first obstacle the beginning PM programmer will encounter is learning the various functions and messages, so as to select the right ones in different situations. Unfortunately there is no easy way to do this; it comes down to experience.

Why Message Passing?

Like other graphical environments, PM supports multiple forms of input to an application program. At least keyboard and mouse are usually present, but pen input is a distinct possibility for the future. Now, consider what might happen if a program - operating in a preemptive multitasking environment like OS/2 - performed its own input processing with two sets of logic for keyboard and mouse. Keyboard processing is relatively simple: just do a table lookup on the input keystroke and branch to the appropriate subroutine. But mouse input is much more complex. We have to deal with selection of objects like blocks of text or even possibly graphics, as well as pull-down menu selections.

It's quite possible that a user might use the mouse to pull down a menu option to save a file, and then immediately press Alt-F4 to quit the application. If the keystroke processing is much faster, then the application might quit before saving the file. Result? One irate user. Of course, most apps would be smart enough to ask the user whether the file should be saved before quitting, but the result would still be annoying and more dangerous problems would likely result.

As a consequence, PM applications have to deal with external events in the order in which they are generated, and to do this PM uses a message-passing architecture. Events, such as mouse movement and button actions, keystrokes and the like, generate messages, which are dispatched to Presentation Manager.

What's a message? Here's how it's defined in the PMWIN.H file:

```
/* QMSG structure */

typedef struct _QMSG { /* qmsg */
     HWND     hwnd;
     ULONG    msg;
     MPARAM   mp1;
     MPARAM   mp2;
     ULONG    time;
     POINTL   ptl;
     ULONG    reserved;
} QMSG;
```

The various members of the structure have the following meanings: hwnd is the handle of the target window. Now, in DOS the only objects which have handles are files, but in OS/2 most things have handles: pipes, queues, semaphores and of course windows. PM will automatically send the message to the appropriate *window procedure* based upon this value. Of course, that window may in turn hand off the message to another window.

The msg field specifies just what the message is. The OS/2 Presentation Manager Programmer's Toolkit header files define several hundred messages. Typical examples would include:

WM_PAINT	Tells the target window that it should repaint its contents on the screen as a result of being opened, brought to the foreground or some other circumstance
WM_COMMAND	The user has chosen a menu option on the application's menu bar or pressed a button.
WM_CONTROL	Something happened to a control window, such as a text-entry field. Just what can be found from the other parts of the message and can range from a user selecting the field with the mouse to typing something into the control.
WM_CHAR	A keystroke.
WM_CLOSE	User is closing the window.

The next two components of the message are MPARAMs. An MPARAM is a 32-bit value which is essentially typeless: it could contain two sixteen-bit values or four eight-bit values or a 32-bit far pointer or almost anything. Generally, programmers break apart an MPARAM, using special macros such as SHORT1FROMMP(), to get at the components. For example, the WM_COMMAND message simply means that the user did something with a menu or some other window component; exactly what is found out by looking at the low word of mp1.

Following the MPARAMS is an unsigned long timestamp which plays a key role in resolving the problem described above. It is the timestamp which keeps messages from being processed in the wrong order. Most applications don't care about the timestamp on a message, and in fact, as you'll see, don't receive the timestamp; as long as the messages are processed in the correct sequence, that's the most important thing.

Lastly, there's a POINTL structure, which contains the x- and y-coordinates of the mouse at the time the message was generated. Again, most programs don't use this use and don't automatically receive it; but it (like the timestamp) can easily be retrieved.

Notice that keystrokes appear to the program as WM_CHAR messages. In fact, PM programs do not read input via C library functions such as gets() and scanf(). Nor do they produce output with puts() or printf(). Instead, all input comes through a message queue and all output is produced by calling PM functions, while the stdin and stdout streams are redirected to the NUL device.

In actual fact, there are two different types of messages: queue messages, which are posted by the system to the application's message queue, and window messages, which are passed as parameters when the system invokes a window procedure. Window messages do not contain the timestamp or mouse position fields.

The main procedure of a PM program will generally receive messages through an input queue, so one of the first things it must do is to create that queue. It will then probably create a window on the screen (not all PM programs do this).

Windows

The application will produce output and receive input through a window. However, not all windows are visible - it is possible to create an 'object window' which will respond to messages and update related data structures without ever appearing on the screen. This corresponds with the idea of an object class in languages like Smalltalk.

In fact, what most users would consider the screen is actually a window - the desktop window (there's also a desktop object window). Windows have relationships in two senses: relative positioning (one window within/on top of another, or behind another) and ownership (one window coordinates the behaviour of another by sending it messages and receiving messages back about the owned window's state). The desktop window is generally the parent for application windows.

The application window the user sees is actually composed of several other windows. In the background is the frame window, which provides a 'base' for several frame controls and the client window area. The frame controls are things like the border, system menu, title bar, minimize/maximize buttons, application menu, icon and scroll bars (if specified) and these are all windows in their own right. These windows are predefined - the code for them is in the operating system itself. The client window is the (usually white) central area where the application displays its output, and the code for this must be written by the programmer.

The frame window coordinates the actions of all the other windows so that they behave according to the OS/2 user-interface guidelines. Move the frame window by dragging the titlebar, and all the other components redraw in the correct locations, for example.

There are other kinds of windows, too. For example, dialog windows are usually used to interact with the user - say, to fill out a form or create a new file - and these windows in turn will contain other control windows such as entry fields, list boxes or pushbuttons. In many cases, the buttons are predefined public classes, although the dialog windows, like client windows, have to be coded by the programmer.

Let's look at a simple OS/2 Presentation Manager application. You'll all be familiar with every C programmer's first C program:

```
#include <stdio.h>

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

Now, the same code can be compiled and run under OS/2, but it would run as a full-screen or windowed character-kernel application. We want the graphical version.

The first job is to include all the definitions of constants and prototypes for the window-manager functions from the Presentation Manager Programmer's Toolkit. Generally, this is done with the preprocessor directive

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

but before this, we specify which subsections of the header files we want by defining some constants such as `INCL_BASE`, `INCL_GPIERRORS` and so on. This is much faster than including all the definitions, which are quite massive.

Next, we need some handles to refer to the various windows. DOS uses handles to identify open files; in OS/2 there are handles for all kinds of objects - files, pipes, queues, and of course, windows. A handle is probably a pointer to some kind of data structure, but you don't need to know - and don't want to know, in case you start to rely on knowledge of the structure and thereby write non-portable code.

Most special types used by PM are typedef'ed in the header files, and PM programmers use these types, even in place of standard C types. The reason is this: in OS/2 1.x, running on a 16-bit processor, an integer is 16 bits in size; but on OS/2 2.x, which runs on 32-bit processors, an integer is 32 bits in size. This is why a message is declared as ULONG above; if we declared it as int then it would be different sizes to different compilers, but a ULONG is always 32-bit.

You'll also notice that OS/2 programmers use Hungarian notation for variable naming. This prefixes the variable name with its type, so that hwndClient is pretty obviously a handle to a window, while szClassName is a string terminated with a zero byte.

```
HWND hwndFrame;
HWND hwndClient;
static char szClassName[] = "Hello";
```

Likewise, hab is a handle to an anchor block, while hmq is a handle to a message queue, and ulFrameFlags is an unsigned long.

The first thing the program must do is register with Presentation Manager. This will cause PM to allocate memory to store graphics images and workspace on behalf of the application. This is done by calling the `WinInitialize()` function, which returns a handle to an anchor block. Nowhere does the PM documentation define an anchor block, but my guess is that it is an instance data segment created by the PM dynamic link library functions. In any case, we don't want to know. From now on, the hab is used as the first parameter for many PM function calls.

Next, the app must create a message queue so that it can read messages. Again, this is a single function call to `WinCreateMsgQueue()`, which returns a handle to the queue. Incidentally, it is this function call which defines a Presentation Manager application. If the program makes this call, it is a PM program; if it does not, it is not, and cannot call most PM functions.

Once this has been done, we can now register a private window class. It's called a class because this one window definition can handle multiple windows, so really a window is an object while the code is a class. Basically, what we are doing is associating a window class name (an English word) with a procedure which will perform input and output processing for all windows of that class. So the parameters to the `WinRegisterClass()` function call are basically the handle to the anchor block, the classname string and a pointer to the window procedure (which is lower in the program code). We also pass some flags which specify class styles, such as redrawing the entire window whenever it is resized. The last parameter is the size of the window data. This is not used in simple applications, but becomes important when an app has multiple windows of the same class. Because a single window procedure must perform processing for multiple windows, it must be reentrant. This means no static variables, and therefore we get the system to reserve a few bytes of memory for each window, and either store each window's private variables along with the window, or store a pointer to a data structure in the window.

Having set all of this up, we can now go ahead and create a window on the screen. Applications can create any type of window with the `WinCreateWindow()` function call, but this requires many parameters and a lot of setting up. In this case, we simply want to create a standard application window, and let the system worry about its size, where on the screen it is placed, and so on.

The `WinCreateStdWindow()` function actually creates (at least) two windows: a frame window and a client window. The client window is the (usually white) area in the center of the application window where work actually gets done. The frame window - only the border of which is visible - is the parent of the client window and other related windows, such as the title bar, system menu, application menu, any scroll bars and so on. The frame window serves to coordinate their bahaviour and paint them in the correct relative positions.

The first thing to do is to specify which component parts we want: border, titlebar, menu, and other on-screen real estate. This is done by setting the component bits of a 32-bit unsigned long. In this case, we want a standard window, except no menu, no accelerator table and no icon:

```
    ulFrameFlags = FCF_STANDARD & ~FCF_MENU & ~FCF_ACCELTABLE &
~FCF_ICON;
```

We could also have OR'ed in bits for `FCF_VERTSCROLL` and `FCF_HORZSCROLL`, if we wanted them. Now comes the actual function call which creates the window. `WinCreateStdWindow()` takes lots of parameters, but fewer than WinCreateWindow()! These include the handle of the parent window (in this case, the desktop), the window style, the frame creation flags, the classname, titlebar text and the address of the variable which will receive the client window handle.

The window styles can include:

`WS_CLIPCHILDREN`	Prevents a window from over-painting its children
`WS_CLIPSIBLINGS`	Prevents a window from over-painting its siblings
`WS_DISABLED`	Disables mouse and keyboard input to the window.
`WS_MAXIMIZED`	Enlarges the window to maximum size
`WS_MINIMIZED`	Reduces the window to smallest (iconized) size
`WS_PARENTCLIP`	Prevents a window from over-painting its parent

`WS_SAVEBITS`	Saves the image in the window as a bitmap, so that if the window is moved or un-hidden, the system can restore the window contents by copying the bitmap without requiring the application to repaint it
`WS_SYNCPAINT`	Causes the system to send a WM_PAINT message to the window whenever any part of it becomes invalid. Without this style, the system accumulates invalid regions and send one WM_PAINT message when no other messages are pending
`WS_VISIBLE`	Makes the window visible (unless it is totally obscured by other windows). Windows without this style are hidden.
`WS_ANIMATE`	Enables a zooming effect on this window (disabled if turned off in the System object in System Setup).

`WinCreateStdWindow()` actually has two parameters of this type: the first specifies the style of the frame window, while the second specifies the style of the client. A frame window can have some additional style flags:

`FS_ACCELTABLE`	Creates an accelerator table
`FS_BORDER`	Creates a window that has an inner border the same color as the titlebar. This style is used by dialog windows
`FS_DLGBORDER`	Creates a window with a single line border (again, a dialog window)
`FS_ICON`	Creates a window with an icon
`FS_MOUSEALIGN`	Creates a window which is positioned relative to the current mouse coordinates. Sometimes used by dialog windows
`FS_NOBYTEALIGN`	Creates a window which is not aligned on a byte boundary in video memory. This will degrade drawing performance.
`FS_SCREENALIGN`	Creates a window which is aligned to the screen (dialog windows)
`FS_SHELLPOSITION`	Allows PM to size the window and place it in a position cascaded from the previously started application window
`FS_SIZEBORDER`	Creates a size border
`FS_SYSMODAL`	Creates a system modal window
`FS_TASKLIST`	Adds the window title to the Window List
`FS_STANDARD`	Combination of FS_ICON, FS_ACCELTABLE, FS_SHELLPOSITION and FS_TASKLIST

At this point in the program, the window gets created and should pop onto the screen. As the window is created, PM sends it various messages, such as `WM_CREATE`, `WM_SIZE` and `WM_PAINT`, to allow the window to initialize, resize itself to the correct size and paint its contents. Of course, the user can also interact with the window, with either the mouse or keyboard (this simple app has almost no keyboard support), and this will cause input to be sent to the app.

As messages arrive on the input queue, we must read these messages and then direct them to the correct window. This is done by the event loop:

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

This reads each message in turn, and dispatches it to the correct window. The process continues until `WinGetMsg()` gets a `WM_QUIT` message, which causes it to return `FALSE` and exit the while loop.

After this, it is all over bar the cleanup, which destroys the window and message queue and releases the anchor block. But where does the actual work get done? We haven't seen any code which prints the message "Hello World".

Window Procedures

Most of the work of PM applications gets done in window procedures, or winprocs, which contain the code which responds to the various messages. A winproc is declared to be of type `MRESULT EXPENTRY`, which means that it returns a 32-bit result code and is called using the `_system` calling convention. This is because it will be called directly by the operating system, not by the application itself (which may use C or OptLink calling conventions). The parameters to the winproc are, of course, the target window handle, the message and its parameters.

The code of most winprocs reduces to a case statement, to handle the different messages, and this one is no exception. Apart from a couple of trivial stubs to handle specific messages, we are really only concerned with one message, `WM_PAINT`. This message is sent whenever the window is created, when it is resized, when it is maximized or brought to the foreground from a partially obscured background position. A really smart application will only redraw that part of its window which was previously invalidated, and it finds that out from the parameters to the `WM_ERASEBACKGROUND` message. This is faster for some applications. However, for simplicity our example ignores `WM_ERASEBACKGROUND` messages and simply redraws the entire window whenever it receives the `WM_PAINT` message.

Here's how it does it:

```
        hps = WinBeginPaint(hwnd, 0, &rc);
```

The `WinBeginPaint()` function call obtains a Presentation Space (PS) and returns a handle to it. Calls to the Presentation Manager GPI (Graphics Programming Interface) functions - and there are hundreds of these - will cause a graphics image to be painted on the screen. One of the nice things about a Presentation Space is that the graphics image can be retained within it and can be output on multiple Device Contexts - that is, a screen window, a printer, a plotter and so on. In this case, the function call obtains a cached micro-PS which is smaller and faster, but can only be used with one Device Context, in this case the screen window. One of the parameters passed to the function is the address of a RECTL (rectangle) structure, rc, which it obligingly fills in with the window coordinates.

Next, we put our message in the window:

```
        WinDrawText(hps, -1, szMessage, &rc, CLR_NEUTRAL,
CLR_BACKGROUND,
             DT_CENTER | DT_VCENTER | DT_ERASERECT);
```

`WinDrawText()` will draw a single line of text in a rectangle in a PS. One of the parameters passed is the address of the RECTL structure containing the rectangle coordinates which we obtained from the WinBeginPaint() call. Others include a pointer to the message string, foreground and background colours and flags to control centering and background erase. Finally

```
        WinEndPaint(hps);
```

ends the painting and releases the PS. That's it.

There are lots of other ways of doing the same thing. For example, I could have used `WinFillRect()` to paint the window white, and then GpiCharStringAt to output the message to the window. And for good measure, I could have added some lines to query the fonts on the system and changed to Times Rmn New, for example.

Of course, other messages also find their way to the winproc, but we don't care about them, so we let the `WinDefWindowProc()` function call PM to perform default message processing.

The complete application is listed below, and includes a few other little twists for error handling.

Real-world apps are of course more complex, but they follow the same basic structure. Bear in mind that because of the need to process messages in strict sequence as described above, the winproc must complete processing one message before it can start another. For this reason, time-consuming message-processing code in one winproc would freeze the system until it was completed. To get around this problem, PM applications will typically start a thread to perform time-consuming tasks such as loading files, while the main thread resumes message processing. This option is not available in Windows, and is the reason why the hourglass icon is so often displayed.

Having seen the amount of code required for this trivial application, you might be asking yourself how much more is involved in a real application. The answer is, not as much as you might think. And if you play with this program, you'll discover it does quite a lot for so few lines of code: it can be moved around the screen, resized, maximized, minimized, sent to the background, brought to the foreground and it has a system menu that allows you to do the same things from the keyboard. It wouldn't be too difficult to turn it into a program that displays a text file, for example. And from there, adding font support and other fancy features is simply a matter of calling the appropriate functions.

And bear in mind that these functions and messages can do a lot. For example, call `WinCreateWindow()` a few times to create some entry-field controls and you have an on-screen form with text editing. Or call `WinCreateWindow()` to create an MLE window: this is a Multiple Line Entryfield which provides all the capabilities of a simple text editor, including cut, copy, paste, wordwrap, tab stops and undo.

One intriguing capability of PM is subclassing windows. This means redefining some of the message-processing characteristics of a window or adding new capabilities while leaving others unchanged. In essence, it means creating a new class of window which inherits some of the properties of its 'parent', in line with the object-oriented programming concept of inheritance.

For a good example of this, take a look at the standard OS/2 system editor, E.EXE. This is simply an MLE field which has been subclassed to add features such as font and color selection, as well as file open and save. It contains embarrassingly few lines of code, although I take my hat off to the programmer for being smart enough to do it this way!

Programming PM tends to be frustrating initially, but becomes more and more rewarding as you master the various function calls and build yourself a library of reuseable routines. Program design is quite different from conventional procedural code in languages such as C, Pascal or dBASE, as there is no neat hierarchy of functions being called from menus and submenus, nor is there a conventional hierarchical input-process-output organisation. A programmer needs to be disciplined to employ good modular coding techniques, hide data behind windows and avoid globals and statics like the plague.

But once the basic concepts are mastered, the result can be some very slick applications indeed.

Listing 1.

```
#define INCL_WIN    /* This includes the window manager function
prototypes, */
             /* constants, etc. */
#include <os2.h>    /* This is the main OS/2 definitions include file
*/

/* Prototype for the applications window procedure */
MRESULT EXPENTRY HelloWndProc(HWND hwnd, ULONG ulMessage, MPARAM mp1,
MPARAM mp2);

/* We need handles for both the frame window and the client window */
HWND hwndFrame;
HWND hwndClient;
/* Multiple windows can be maintained by one wndproc, so it's a
class, rather
than an object. The class is registered by name with the system */
static char szClassName[] = "Hello";

void main()
{
    HAB hab;       /* Handle to an 'anchor block' */
    HMQ hmq;       /* Handle to the input message queue */
    QMSG qmsg;     /* Queue message structure to hold the incoming
message */
    ULONG ulFrameFlags;   /* Frame creation flags */

    /* First, register with PM to get services. This returns a handle
to an anchor block */
    hab = WinInitialize(0);
    /* Next, create a message queue */
    hmq = WinCreateMsgQueue(hab, 0);
    /* Now register a window class for the application */
    if(!WinRegisterClass(hab,
         (PCH)szClassName,
         (PFNWP)HelloWndProc,    /* Pointer to winproc function */
         CS_SYNCPAINT | CS_SIZEREDRAW,   /* Flags to force redraw
when resized, etc. */
         0))                /* Number of bytes in 'window words' */
     DosExit(EXIT_PROCESS, 1);

    /* We want a regular window, but without a menu, accelerator
table and icon */
    ulFrameFlags = FCF_STANDARD & ~FCF_MENU & ~FCF_ACCELTABLE &
~FCF_ICON;

    /* Now go ahead and create the window */
    /* This function call returns the frame window handle and also
sets the client window handle */
    hwndFrame = WinCreateStdWindow(HWND_DESKTOP,    /* Parent window
*/
             WS_VISIBLE,             /* Window style visible */
             &ulFrameFlags,            /* pointer to frame flags */
             (PCH)szClassName,            /* Registered class name */
             "Hello",                /* Text for title bar */
             WS_VISIBLE,             /* Client window style */
             (HMODULE)NULL,          /* Pointer to resource module */
             0,                      /* Resource ID within module */
             (HWND *)&hwndClient);        /* Pointer to client window
handle */
```

```
    if(hwndFrame == 0) {
        WinAlarm(HWND_DESKTOP, WA_ERROR);
        DosExit(EXIT_PROCESS, 1);
    }

    /* Now loop around processing events. This is called the 'event
loop' */
    /* WinGetMsg returns FALSE when it gets the WM_QUIT message */
    while(WinGetMsg(hab, &qmsg, 0, 0, 0))  /* Get a message */
        WinDispatchMsg(hab, &qmsg);        /* Dispatch it to the
window */

    /* Lastly, clean up */
    WinDestroyWindow(hwndFrame);
    WinDestroyMsgQueue(hmq);
    WinTerminate(hab);
}

/* The client window procedure. Gets called with message components
as parameters */
MRESULT EXPENTRY HelloWndProc(HWND hwnd, ULONG ulMessage, MPARAM mp1,
MPARAM mp2)
{
    HPS hps;      /* Handle to a presentation space. This is where a
PM program 'draws' */
    RECTL rc;     /* A rectangle structure, used to store the window
coordinates */
    CHAR szMessage[] = "Hello World!";     /* The message we'll
display */

    switch(ulMessage) {

     case WM_CREATE:     /* process this message by returning FALSE.
This lets the */
         return (MRESULT)FALSE;  /* system continue creating the
window */
         break;

     case WM_ERASEBACKGROUND:    /* Let the frame window procedure
redraw the background */
         return (MRESULT)TRUE;   /* in SYSCLR_WINDOW (usually white)
*/
         break;

     case WM_PAINT:    /* The 'guts' of the application */
         hps = WinBeginPaint(hwnd, 0, &rc );   /* Get a presentation
space */
        WinQueryWindowRect(hwnd, &rc);
         /* Draw the message, in rectangle rc, coloured CLR_NEUTRAL
(black) on
         CLR_BACKGROUND (white), centered, over-writing the entire
rectangle */
         WinDrawText(hps, -1, szMessage, &rc, CLR_NEUTRAL,
CLR_BACKGROUND,
               DT_CENTER | DT_VCENTER | DT_ERASERECT);
         WinEndPaint(hps);       /* Release the presentation space */
         break;

     case WM_CLOSE:/* User chose CLOSE on system menu or double-
clicked */
```

```
         WinPostMsg(hwnd, WM_QUIT, 0, 0);  /* So send back WM_QUIT,
causing */
         break;            /* WinGetMsg to return FALSE and exit
the event loop */

     default:     /* Let the system handle messages we don't */
         return WinDefWindowProc(hwnd, ulMessage, mp1, mp2);
         break;
    }
    return 0L;
}
```

Basic Window Concepts

In this section, I want to introduce more detail on some of the ideas previously introduced and emphasise some basic concepts.

Window Procedure Structure

In almost all cases, the window procedure is basically a gigantic `switch()` statement. Not always, however: in the simple example above, I could actually have written it as an `if()` statement. This is because the default window procedure `WinDefWindowProc()` actually provides exactly the behaviour I wrote for `WM_CREATE`, `WM_ERASEBACKGROUND` and `WM_CLOSE`. So actually, I could have written:

```
if (ulMessage == WM_PAINT) {
    /* Do the painting here */
}
else return WinDefWindowProc(hwnd, ulMessage, mp1, mp2);
```

However, that wouldn't have been a particularly representative-looking window procedure, which would have defeated the point of the exercise.

In many programs, we take advantage of the scoping features of the C language to define variables in the stubs of the `switch()` statement:

```
case WM_PAINT: {
    HPS hps;
    RECTL rc;

    hps = WinBeginPaint(hwnd, 0, &rc);
    .
    /* (etc) */
    .
    WinEndPaint(hps);
    return (MRESULT) FALSE;
}
```

In more complex programs, the stubs of the switch() statement become quite long, and so to avoid losing track of what belongs where, as well as indenting code off the right-hand side of the page, we instead call functions:

```
case WM_PAINT:
    return paint(hwnd);
```

In this example, the only parameter passed to the paint() function is the hwnd. This is because there is no point in passing the message - we already know it was WM_PAINT or we wouldn't be calling the paint() function - and the WM_PAINT message passes no other information in the MPARAMs.

The paint() function would look like this:

```
MRESULT paint(HWND hwnd)
{
    HPS hps;
    RECTL rc;

    hps = WinBeginPaint(hwnd, 0, &rc);
    .
    /* (etc) */
    return (MRESULT) FALSE;
}
```

Notice that its return type is the same as the window procedure.

This layout helps to keep the window procedure down to a manageable size so that we can see its overall logic, and all the messages handled, on a single page or screen.

More complex structures are possible, such as making message despatching be table-driven or even implementing a finite-state machine, but these approaches are rarely justified.

Generally, a window procedure is prototyped in the following canonical form:

```
MRESULT EXPENTRY wpType(HWND hwnd, ULONG msg, MPARAM mp1, MPARAM mp2)
```

Why is the window handle, `HWND hwnd`, passed in as a parameter? The answer is that a single window procedure or winproc can operate on behalf of more than one window. For example, look at your screen right now - there are several application windows open, each with a sizeborder (visible part of the frame), titlebar, sysmenu, minmax and so on. Now, it would be very inefficient to have a separate window procedure for every titlebar, so one winproc does them all. However, when it is called, it needs to know which titlebar it should change the text in, or highlight, or repaint, or whatever.

Inside the window procedure, you should read the variable `hwnd` as this window. It's a bit like the `this` pointer in C++ programming, or like `SOMself->` in SOM programming.

Window ID's and Window Handles

What is a handle? Good question - sometimes a handle is a pointer to some kind of control structure. Sometimes it is an index into an array of pointers to control structures (that's what DOS file handles are). But the correct answer to this question is 'who knows? - and better yet, who cares?'.

Different systems (operating systems, language compilers and even hardware architectures) implement handles in different ways, and we don't want to know what a handle is, as this may trap us into writing non-portable code. However, there is one thing we can say for sure about handles: they are run-time values which are allocated by the operating system from internal tables or other structures. Right now, the frame window of my word processor has a particular window handle value. If I exit it and restart it, it will have a different frame window handle.

We cannot know handle values at compile time, so as we write our code we cannot simply make them up. Handles are always assigned by the system, as the returned value of API's like `WinCreateWindow()`, `WinCreateStdWindow()`, `DosOpen()` and others.

But how, then, can we distinguish between different windows? We know the client window handle, and we know the frame window handle. But what if our application client window then displays two static controls, two entryfields and three pushbuttons - how can we manipulate those windows if we do not know their handles? Virtually all of the PM API's use window handles to identify the control being dealt with.

The answer lies in window ID's. These are integral values, which we identify by symbolic constants. For example, in an application shared header (.H) file, we might write:

```
#define IDEF_HEXVALUE  0x0501
#define IDEF_DECVALUE  0x0502
#define IDSS_HEXPROMPT 0x0503
#define IDSS_DECPROMPT 0x0504
#define IDPB_CONVERT   0x0505
#define IDPB_CLEAR     0x0506
#define IDPB_CANCEL    0x0507
```

In this case, I use a naming convention: ID (for ID, obviously); EF for entryfield, SS for static control and PB for pushbutton, followed by the symbolic name of the control.

Now, when I want to get the window handle, I can call an API: `WinWindowFromID()`. This API takes two arguments: the parent window handle and the window ID of the child. Since it is generally used inside a window procedure to get the handle of an immediate child of this window, the invocation would typically look like:

```
hwndEFHex = WinWindowFromID(hwnd, IDEF_HEXVALUE);
```

Remember, in a window procedure, `hwnd` is read as *this window*.

Another useful API is `WinQueryWindow()` which is used when you know the handle of one window (like `hwnd`) and want the handle of a window which is related to it in some way - like a parent, or owner, or previous or next in the screen Z-order. An example might be

```
hwndMyParent = WinQueryWindow(hwnd, QW_PARENT);
```

As a general rule, it is best to work out window handles from the window relationships at runtime, rather than using global variables with names like `hwndMainWindow` or `hwndParentForm`. This technique allows for better modularity and code reuse. Your goal should be to write completely independent window classes which can be reused in different applications.

Window Creation

It is important to realise that almost everything that exhibits any kind of behaviour in Presentation Manager is a window. Pushbuttons, menus, containers, notebooks, static controls, scrollbars, entryfields, MLE's, valuesets - they're all windows. They all have a window procedure (or window class) provided in the operating system, and they are all created in the same way.

Window creation is done by the `WinCreateWindow()` API. A typical call, showing the parameters passed to it might be:

```
hwndFrame = WinCreateWindow(
     HWND_DESKTOP,       /* Parent window */
     WC_FRAME,           /* Window Class */
     (PSZ)NULL,          /* Window Text */
     0L,                 /* Window Styles */
     0,                  /* Bottom Left x */
     0,                  /* Bottom Left y */
     0,                  /* Width */
     0,                  /* Height */
     (HWND) 0,           /* Owner window handle */
     HWND_TOP,           /* Z-order */
     ID_MAIN,            /* Window ID */
     &fcData,            /* Creation Data */
     NULL);              /* PRES PARAMS */
```

Running through this call from top to bottom:

The returned value of the call is the new window handle.

The first parameter is the handle of its parent. Windows are always positioned relative to the botton-left corner (0,0) of their parents, and they are generally clipped to appear within their parents, too. Parentage controls on-screen positioning and appearance.

The next parameter is the window class - in other words, what type of window this will be.

Next comes the window text, if there is any. Titlebars, static text controls and entryfields all display text; some other classes do not.

Then come the window style bits, which have been discussed before.

The next four parameters set the window's initial position (x, y) and size (cx, cy). In the example above, the position and size are all zero, and the window does not have style `WS_VISIBLE`. This is because it is a frame window, and in the application logic, the next thing that will happen is creation of the client window, and it it turn will create its children (statics, entryfields, pushbuttons, etc). If this is done with the windows visible, the result will be that the user will see the assembly of the application window and the resizing of the controls, particularly on a slow machine. A better way to do things is to create and size the windows from top to bottom, and then make them all visible. Voila!

Next we have the owner window handle. Ownership (and parentage) are discussed in a later section - for now, suffice it to say that ownership is a way of specifying which window coordinates and responds to the actions of this window. In this case, since this is a frame window, it does not have - and does not need - an owner window.

Next, the Z-order. If left-right is the x dimension and up-down is y, then the Z-axis is front-to-back overlaying of windows on the screen. Since this is a new application window which the user has just launched, it had better be in front so the user can start using it immediately, hence the value of `HWND_TOP`.

Next, the window ID, as discussed above.

Second-to-last, we pass in the address of a creation structure. Just what this structure is depends upon the class of the window being created. In this case, it is a `FRAMECDATA` structure, which allows us to specify the frame creation flags, the DLL module handle from which to obtain resources and the default resource ID.

Creation of a different window class would require a different creation structure. And if you are creating your own window classes (and you are!), the creation data structure is a great window of passing initialisation data, or parameters, to your window, rather like passing parameters to a function. Remember, PM programming involves decomposition of the application into windows, not functional decomposition.

Lastly, we pass in a pointer to a `PRESPARAM` structure. Presentation parameters allow the programmer to over-ride the default colours and fonts for a window and the windows it owns.

PM's Message-Passing Architecture

As we now know, Presentation Manager is a message-passing system. It is designed that way in order to provide consistent type-ahead and a predictable sequence of processing for input. Take a look at Figure 1.

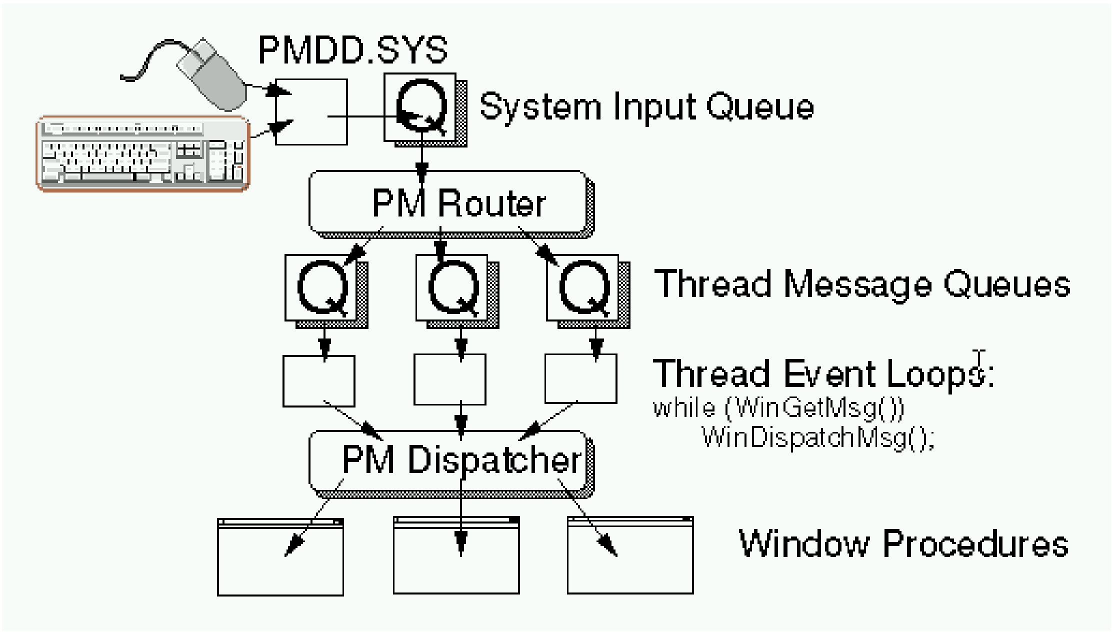


Figure 1. Message flow through the Presentation Manager architecture

All input, whether from keyboard, mouse or other input devices such as touch-sensitive screens and digitizers, is passed from the physical device driver to the PMDD.SYS device driver, which converts it into a message. This includes mouse movement, mouse button actions (down, up) and key events (key down, key up). These messages are placed on the system input queue.

From there, the PM router places the messages on the corresponding application queues. For simple applications, which are usually single-threaded, there is a single queue per application. However, more sophisticated applications are often multithreaded, and in that case will have a message queue for many of those threads (but not necessarily all - a thread does not have to have a message queue and in some circumstances it is a positive drawback).

The application/thread message queues are not embedded in the system; rather they are created in the anchor block for each thread, by calling the WinCreateMsgQueue() API. Messages are picked off the queues by the threads calling WinGetMsg(), and usually (but not always) then dispatched to the corresponding winproc by calling WinDispatchMsg().

It is important to distinguish conceptually between two types of messages:

- the messages which are sent to your window procedure by the system, perhaps as a result of the user manipulating a window or as a response to some system event. These messages must be dealt with in your window procedure, and

- the messages which you send to yourself or other windows in order to make them perform some function.

A good example might be the `WM_QUIT` and `WM_CLOSE` messages, which often cause confusion for novice PM programmers. `WM_CLOSE` is an example of the first type of message: it is sent by the system (actually, the frame window) when the user double-clicks on the system menu or otherwise closes the window. In response to this message, a typical strategy is to use the `WinMessageBox()` API to ask the user if he really wants to quit or wants to save work, and then perform termination processing. This processing will usually end with the winproc posting `WM_QUIT` back to itself in order to terminate the event loop. The confusion arises because, if `WM_CLOSE` is not explicitly handled, it passes through the default stub and is dealt with by `WinDefWindowProc()`, which posts `WM_QUIT` back to the window anyway.

Packer & Cracker Macros

A window procedure is called by the PM Dispatcher to handle a message for a window, and is passed four parameters: the window handle, the message itself, and two MPARAMs, which contain message parameters. Just what these message parameters represent depends upon the message. In some circumstances, an MPARAM might contain a pointer to a structure, at other times it might contain a window handle, the index of an item in a listbox, two unsigned shorts which represent indexes, window ID's or even four unsigned char fields which are bitfields containing status information.

We need some way of pulling apart an MPARAM. Furthermore, it is unwise to simply attack the MPARAM by left- or right-shifting and ANDing to mask out the bits we want - this approach will fail when we port our code to other architectures, due to variations in word size, little- or big-endedness of the processor and other factors.

For this reason, the header files - actually, PMWIN.H - provide a number of macros which will implement the desired functionality on different platforms. These macros are commonly known as the 'packer and cracker' macros.

First, the crackers. Inside a winproc, you are passed two MPARAMs and need to pull them apart. The macros which do that are:

```
PVOIDFROMMP(mp)
HWNDFROMMP(mp)
CHAR1FROMMP(mp)
CHAR2FROMMP(mp)
CHAR3FROMMP(mp)
CHAR4FROMMP(mp)
SHORT1FROMMP(mp)
SHORT2FROMMP(mp)
LONGFROMMP(mp)
```

So, for example, in processing a `WM_COMMAND` message, you wanto to know what the menu choice is. The online help tells you that mp1 contains a USHORT which is the command, so to extract it, you could write

```
usMenuChoice = SHORT1FROMMP(mp1);
```

Then you might want to know whether it was produced by a menuitem, a pushbutton or an accelerator table. This information is in short 1 of mp2, so

```
usCmdSrc = SHORT1FROMMP(mp2);
```

gets that information. Or suppose you receive a `WM_CONTROL` message in a dialog. You need to work out, first what control it came from, and then, what the event was. The window ID of the control is in short 1 of mp1, while the notification code that tells you what happened is in short 2 of mp1. Thus:

```
case WM_COMMAND: {
    switch(SHORT1FROMMP(mp1)) {    /* Identify the control */
        case IDEF_TITLE:
            switch (SHORT2FROMMP(mp1)) {
                case EN_KILLFOCUS: // User moved off control, read it
                    .
                    break;
                case EN_CHANGED:   // Field is dirty
                    .
                    break;
            }
        case IDEF_FIRST:
            switch (SHORT2FROMMP(mp1)) {
                case EN_KILLFOCUS: // User moved off control, read it
                    .
                    break;
                case EN_CHANGED:   // Field is dirty
                    .
                    break;
            }
    }
}
```

In this case, I've chosen to switch on the control ID first, and then the notification code. But if you have lots of identical controls - e.g. a form with lots of entryfields, all with identical behaviour, you may rearrange the logic to switch on the notification code, and then switch on the control ID or use it as an index into an array of structures which represent the data behind the form.

There's another set of crackers which are useful. These are used when you send a message to another window and want to make sense of the returned value, which is an MRESULT (message result):

```
PVOIDFROMMR(mr)
SHORT1FROMMR(mr)
SHORT2FROMMR(mr)
LONGFROMMR(mr)
```

For example, suppose you have written a window class which plots items at certain x/y coordinates, and you have defined a message to which the window responds with the x and y coordinates, given the item number:

```
MRESULT coord;
USHORT itemnumber;
for(itemnumber = 0; itemnumber < 5; itemnumber++) {
    coord = WinSendMsg(hwndGraph, UM_QUERYCOORDS, itemnumber, 0L);
    x[itemnumber] = SHORT1FROMMR(coord);
    y[itemnumber] = SHORT2FROMMR(coord);
}
```

Now, to the packers. How could the window procedure in this example have packed the x and y coordinates into a single returned value? The packer macros which generate a message result are:

```
MRFROMP(p)
MRFROMSHORT(s)
MRFROM2SHORT(s1, s2)
MRFROMLONG(l)
```

So, in the code for this hypothetical window class, the UM_QUERYCOORDS message processing stub would end with:

```
        return MRFROM2SHORT(x,y);
        break;
```

Similarly, there are packers for assembling MPARAMs:

```
MPFROMP(p)
MPFROMHWND(hwnd)
MPFROMCHAR(ch)
MPFROMSHORT(s)
MPFROM2SHORT(s1, s2)
MPFROMSH2CH(s, uch1, uch2)
MPFROMLONG(l)
```

Common Useful Messages

`WM_CREATE`

`WM_DESTROY`

`WM_QUIT`

`WM_PAINT`

`WM_COMMAND`

`WM_CONTROL`

`WM_SIZE`

`WM_CHAR`

The `WM_CHAR` message is used to send a keystroke to a window. Notice that this is quite an unusual circumstance in most business applications, as generally the client window owns a number of entryfields and similar controls and one of these has focus and receives keystrokes so that the client window procedure never sees them. However, an entryfield will process most keystrokes but not, for example, tab, backtab, enter and others. Internally, the entryfield passes unprocessed `WM_CHAR` messages to `WinDefWindowProc()`, which will pass them to the entryfield's owner, so that in this case, the client window will see these keystrokes as `WM_CHAR` messages.

If the entryfield's parent is a dialog window, the dialog window procedure will automatically (via `WinDefDlgProc()`) process the tab key by setting the keyboard focus to the next control which has `WS_TABSTOP` windowstyle.

The system generates `WM_CHAR` messages for keyboard state changes, so that for a user keystroke, at least two `WM_CHAR` messages will be generated: one for key down, and one for key up. If the user holds the key down so that the key auto-repeats, then it will generate multiple WM_CHAR messages for the key-down event, and if this occurs faster than the thread can read the messages from its input queue, then the system will combine multiple WM_CHAR messages into a single message with a count byte set to greater than one.

The `WM_CHAR` message is decoded by examining the status bits in short 1 of mp1

The simplest way to extract the various fields from the two MPARAMs of a `WM_CHAR` message is to use the `CHARMSG(&msg)` macro. *Warning*: this macro relies for its operation on the fact that the four parameters passed to a window procedure are adjacent to each other in a certain order on the stack. This may change on other architectures, so flag any code that uses this macro as a possible target for trouble when porting. Having said that, we were warned in the days of OS/2 1.x 16-bit code that it could prove troublesome in future releases, yet we're still using it in Warp.

The `CHRMSG` struct and the `CHARMSG()` macro are defined as follows (in PMWIN.H):

```
typedef struct _CHARMSG    /* charmsg */
{
   USHORT  fs;           /* mp1     */
   UCHAR   cRepeat;
   UCHAR   scancode;
   USHORT  chr;          /* mp2     */
   USHORT  vkey;
} CHRMSG;
typedef CHRMSG *PCHRMSG;

#define CHARMSG(pmsg) \
((PCHRMSG)((PBYTE)pmsg + sizeof(MPARAM) ))
```

Notice that the argument to the `CHARMSG()` macro is the address of the msg parameter passed to the winproc. Here is an example of its use to ignore key releases:

```
    case WM_CHAR:
        if(CHARMSG(&ulMessage)->fs & KC_KEYUP) // Ignore key releases
            return (MRESULT) FALSE;
            break;
        switch(CHARMSG(&ulMessage)->vkey) {
            case VK_TAB: // Move to next entryfield
                .
                break;
            case VK_BACKTAB: // Move to previous entryfield
                .
                break;
        }
        return (MRESULT) TRUE; // Indicate keystroke processed
        break;
```

This little fragment also shows the processing of virtual key codes. Virtual keys are the keys on the PC keyboard that do not generate an ANSI (ASCII) code, such as F1 - F24, NumLock, Insert, PageUp, SysRq and so on. The VK_ keys also include tab, backtab, Ctrl, shift, newline and some others.

The `CHARMSG(&msg)->fs` status bits include `KC_ALT` (Alt key was pressed when the message was generated), `KC_SHIFT` (Shift key was down), `KC_CTRL` (Ctrl key was down), `KC_CHAR` (the message contains a valid character code, usually ASCII), `KC_VIRTUALKEY` (message contains a valid virtual key code) and `KC_TOGGLE` (which toggles on and off for keys like NumLock). Some of the other status bits are used in connection with composition of characters with diacritical marks

Message Processing - Sending vs Posting

Menus

Usually the first capability we need to add to simple programs for user interaction is menu support.

In a conventional procedural programming environment, using a language like xBase, COBOL or whatever, we would code a simple full-screen menu like this:

```
main procedure:
  more = true
  do while more
    clear screen and display menu choices
    getkeystroke into choice
    select on choice
      case 1: call proc1
      case 2: call proc2
      case 3: call proc3
      case Q: more = false
    end select
  end do
```

or similar. Of course, any of the subprocedures 1, 2 or 3 could equally well display a menu in the same manner, allowing for a more sophisticated tree of submenus.

The major problem with this scheme is that it is limiting for the user. At any moment, the user can only choose from the menu that is before him. In effect, we are forcing the program logic to restrict the choices available to the user, and the result is that the user feels the program is not flexible enough, or not powerful enough. It is frustrating and not at all empowering.

In addition, the menus can take up quite a lot of screen area, and it becomes difficult to display lots of options.

Under OS/2 Presentation Manager, menu display is quite different. The application contains little, or more often, no code relating to menu display. Instead, all that is taken care of by Presentation Manager internally. The application only contains code to respond to the user's menu selections.

What Is A Menu?

A menu is actually a tree of menus, submenus and related menuitems, and each menuitem is actually a window. The menuitems are not normally visible; in fact the system hides them until they are needed by making them children of `HWND_OBJECT`, and then reveals them by switching the parent to be the menu.

The simplest way to create a menu is by defining it as an application resource. This is done by writing a resource compiler script and running it through the resource compiler, RC.EXE.

The Resouce Compiler, and its script language, introduces a new set of required skills for the Presentation Manager programmer. It is no longer enough to write procedural code; now the application consists of a tree of related components, mostly windows, which are displayed on-screen, with procedural code to define their interactions. However, the simplest way to create the components is not through C code, but instead by loading them as resources.

In short, you could think of the C code you write as being the verbs of your program, and the resources as the nouns. The resources are entities such as windows, dialogs, strings and messages, bitmaps, icons and menus. The C code breathes life into them.

There are a number of benefits to this separation. The first is that it takes a lot of the effort out of coding. Laying out a complex dialog, for example, will involve a lot of code to create the various windows and controls and lay them out correctly, whereas the Dialog Editor and Resource Compiler reduce this to a kindergarten painting-by-numbers exercise.

Secondly, the elimination of all button text, menu text and other strings from the executable source makes the program easier to port for international markets. Separate resource files can be generated for English, French, German, Italian and other national-language versions, and the executables generated without altering the source code. Unfortunately, support for double-byte character sets is not so easy to add, so that the vast S.E. Asian markets might as well be on the moon as far as many developers are concerned.

Resource Compiler Files

Here is a simple example of a resource script which includes an icon, four bitmaps and a menu:

```
#define INCL_WIN
#include <os2.h>

#include "draw.h"

ICON ID_DRAW "draw.ico"

BITMAP ID_BRUSH brush.bmp
BITMAP ID_LINE line.bmp
BITMAP ID_SQUARE square.bmp
BITMAP ID_CIRCLE circle.bmp

MENU ID_DRAW PRELOAD
BEGIN
    SUBMENU "~File", IDM_FILE
    BEGIN
        MENUITEM "~New", IDM_NEW
        MENUITEM "~Open", IDM_OPEN
        MENUITEM "~Save", IDM_SAVE
        MENUITEM "Save ~As", IDM_SAVEAS
        MENUITEM SEPARATOR
        MENUITEM "E~xit", IDM_EXIT
    END
    SUBMENU "~Edit", IDM_EDIT
    BEGIN
        MENUITEM "~Cut", IDM_CUT
        MENUITEM "C~opy", IDM_COPY
        MENUITEM "~Paste", IDM_PASTE
    END
END
```

The first thing you will notice is that it starts by `#defining INCL_WIN` and then `#includes <os2.h>`. This is because the resource compiler is able to apply certain menuitem styles and attributes (as well as more general window styles), and these are defined as constants in the system header files, so that if they are not included, the result will be a compile-time error.

In addition, it #includes the project header file, in this case "`draw.h`". This is because the menuitems are actually windows, and their window ID is sent as part of the `WM_COMMAND` message which signals to the window procedure that a menuitem has been chosen. In response to the `WM_COMMAND` message, the window procedure runs a `switch` statement, in which each case stub corresponds to a menuitem ID. Since the two files must agree on what the ID's are, the ID's are defined in a shared header file, and included into the various source (.C, .RC) files which need it. Here is an extract from the .H file:

```
#define WC_DRAW     "Draw"
#define WC_CANVAS   "Canvas"

#define APP_TITLE   "Draw"

#define ID_DRAW     0x0100

#define IDM_FILE    0x0101
#define IDM_OPEN    0x0102
#define IDM_NEW     0x0103
#define IDM_SAVE    0x0104
#define IDM_SAVEAS  0x0105
#define IDM_EDIT    0x0106
#define IDM_CUT     0x0107
#define IDM_COPY    0x0108
#define IDM_PASTE   0x0109
#define IDM_EXIT    0x010A

#define ID_TOOLBAR  0x0200

#define ID_BRUSH    0x0001
#define ID_LINE     0x0002
#define ID_SQUARE   0x0003
#define ID_CIRCLE   0x0004
```

Notice that window (and menu) ID's are simply integers, but we give them symbolic names to make it easier to keep track of them in our code. Now, returning to the resource compiler script, it starts with:

```
MENU ID_DRAW PRELOAD
```

This line starts the menu definition, and specifies the menu ID. This is important. When the application frame window is created, if the Frame Creation Flags specify `FCF_MENU` then the `WinCreateStdWindow()` function call will create a frame and that frame will want to load the application menu from the resources, but the question is - which menu? There can be several menus in the resources. Of course, they are distinguished by their ID's, and the answer is that the ID of the default menu must match the resource ID passed as an argument to `WinCreateStdWindow()`. A further complication is that the Frame Creation Flags may also call for a default icon and a default accelerator table, and these must all have the same ID.

Following the menu ID, we can specify some options, such as when we want the menu loaded; this can be `PRELOAD` (load when the application starts) or LOADONCALL (load when the application calls `WinLoadMenu()` - this is the default).

The menu definition itself is enclosed within the following `BEGIN` and `END` statements. If you prefer, these can be replaced by curly braces.

Now comes the definition of each submenu. Each `SUBMENU` statement is followed by the submenu text, in quotes, and the submenu ID. A tilde (~) before a character causes it to be underscored, and the user can now select that submenu by using the Alt key and appropriate character. This works for menuitems also.

The submenu definition is also bracketed by `BEGIN` and `END` statements. Submenus can include other submenus - the menus just cascade out sideways when selected, but sooner or later you must end up with some `MENUITEM` statements. The basic reason for this is that only when the user finally selects a `MENUITEM` does the system send a `WM_COMMAND` message to the owner window, thereby causing something to happen. Your application window procedures will never see a `WM_COMMAND` message with a window ID for the `MENU` or `SUBMENU` items as they can never be selected by the user (actually, other messages relating to advanced menu operation are sent by the menu and submenus, but they are dealt with in the course OS291 - Advanced OS/2 Warp & Presentation Manager Programming).

Each menuitem, and also each submenu, can additionally have menuitem styles applied to it. These allow the programmer to define a menuitem as being a bitmap rather than text, as being a help menuitem (so it sends `WM_HELP` rather than `WM_COMMAND`), as being a system menu item (generates `WM_SYSCOMMAND`) and so on. In addition, menuitems can also have menuitem attributes, such as being checked, being disabled and so on. (These are also dealt with in OS291 - Advanced OS/2 Warp & PM Programming).

Dealing with Menu Messages

When the menu is displayed at runtime, it is managed entirely by Presentation Manager. Some messages relating to advanced options are sent to your application's client window procedure, but if you choose to ignore them `WinDefWindowProc()` will automatically provide default behaviour. The only message you need deal with is `WM_COMMAND`, which indicates that the user has made a menu selection.

The `WM_COMMAND` message has three arguments which are packed into its two MPARAMs, and we must use the cracker macros, or casting, to extract them. The most important is the window ID (a.k.a. the menuitem ID) of the menuitem selected, which is short 1 in mp1. For most applications, that is all we care about. We don't care whether the user used the mouse or keyboard, or used a Ctrl-key accelerator. So, the `WM_COMMAND` processing typically looks like this:

```
switch(msg) {
    case WM_CREATE: /* Blah blah */
        return (MRESULT) FALSE;
        break;
    /* Other message cases go here */
    case WM_COMMAND:
        switch(SHORT1FROMMP(mp1)) {
            case IDM_NEW:
                /* do the 'new document' stuff */
                return (MRESULT) FALSE;
                break;
            case IDM_OPEN:
                /* display file open dialog and load file */
                return (MRESULT) FALSE;
                break;
            case IDM_SAVE:
                /* save the file */
                return (MRESULT) FALSE;
                break;
            case IDM_SAVEAS:
                /* display 'save as' dialog and save file */
                return (MRESULT) FALSE;
                break;
            /* Other menuitem cases go here */
            default:
                /* Don't know what's going on here */
                WinMessageBox(HWND_DESKTOP, HWND_DESKTOP, "Unexpected
command", "Error!", 100, MB_EXCLAMATION);
                return (MRESULT) FALSE;
                break;
        }
        /* Other message cases go here */
    default:
        return WinDefWindowProc(hwnd, msg, mp1, mp2);
        break;
}
```

Notice that there is no other hierarchical organization of the menu-processing code in the application. Essentially, any menu selection can occur at any time, so if certain selections are inappropriate, it is up to the programmer to disable them (using menuitem attribute `MIA_DISABLED`).

Window Words

Beginning Presentation Manager programmers often wonder about the terminology and style of PM programs. Why, for example, does a program have to register a window class before creating a standard window? You often read comments about creating a window using the window's registered class - what does that mean? And does this relate to the use of classes in object-oriented programming?

It's possible to create quite complex programs in PM without dealing with these problems directly. However, sooner or later, they have to be confronted. And that's what this article is about.

As you run multiple applications under PM, you'll notice that certain window components appear all over your screen. There are multiple title bars, for example, and minimise/maximise icons, and menu bars, and system menu icons. At any one time, a dialog window might display multiple text entry fields, multiple list boxes and multiple push buttons.

Think about this for a moment. As you may know, the different window components and controls in OS/2 are in fact windows in their own right (unlike in Windows!). That means that they must have their own winprocs, right?

Now, take a look at some of your own winprocs. Or, take a look at some of the examples in the Petzold book. Notice anything about the variables used in the winproc code? A lot of them are static, which means that the winproc is not reentrant - it can only be used by a single window. For example, suppose the window colour was stored in a static variable in the winproc (as it is in some of Petzold's code). Then, if two windows of the same type were created, they would both have the same colour, and if it was changed for one, it would also change for the other - which may or may not be the intended effect. Much worse effects are likely - they're called bugs.

Clearly, the various different title bars currently on your screen have different coordinates, sizes, colours, text and so on. Those attributes cannot be stored in static variables in the winproc, so where are they? The answer lies in *window words*, also known as *instance data*.

The `WinRegisterClass()` function call takes five parameters: the anchor block handle, pointer to the window class name, pointer to the winproc, a class style and the number of bytes of storage reserved in each window of this class. It is this last parameter that is of most interest to us.

Once the window class has been registered, whenever a program calls `WinCreateStdWindow()` to create a new window of this class, the system creates the appropriate data structure internally which represents this window. Just what is in this structure is private to the system, and writing applications which depend upon a knowledge of the structure is asking for trouble in the form of future portability problems. However, an application is able to access any of the following fields:

```
pointer to window-class data structure         WinQueryClassName(),
                                               WinQueryClassInfo()
pointer to window procedure                    WinQueryWindowPtr(hwnd,QWP_PFNWP)
parent-window handle                           WinQueryWindow(hwnd,QW_PARENT,flock)
owner-window handle                            WinQueryWindow(hwnd,QW_OWNER, flock)
owner of this window's frame
                                                   WinQueryWindow(hwnd,QW_FRAMEOWNER,
                                               flock)
handle of the topmost child window             WinQueryWindow(hwnd,QW_TOP,flock)
handle of the bottom child window              WinQueryWindow(hwnd,QW_BOTTOM,flock)
```

`handle of the next sibling window`	`WinQueryWindow(hwnd,QW_NEXTTOP,flock)`
`handle of the previous sibling window`	`WinQueryWindow(hwnd,QW_PREVTOP,flock)`
`handle of the window below in z order`	`WinQueryWindow(hwnd,QW_NEXT,flock)`
`handle of the window above in z order`	`WinQueryWindow(hwnd,QW_PREV,flock)`
`window size and position (as _SWP structure)`	`WinQueryWindowPos()`
`window size and position (as _RECTL structure)`	`WinQueryWindowRect`
`lock count`	`WinQueryLockCount()`
`window style`	`WinQueryWindowULong(hwnd,QWL_STYLE)`
`update-region handle`	`WinQueryUpdateRect(),` `WinQueryUpdateRegion()`

The `flock` parameter is required in OS/2 1.x (where it is ignored in 1.2 and later), but is not required in OS/2 2.x.

When a non-zero value is specified for that last parameter to WinRegisterClass() subsequent calls to WinCreateWindow or WinCreateStdWindow will not only create the usual window structure, but also append the specified number of bytes into the end of the structure, for use by the application programmer. Notice that this data structure is created for each window.

Setting and Querying Values

The programmer can now use these bytes - which are referred to as *window words* - to store value that are associated with that particular window in the window structure. The winproc relates to a *class*, while the window structure relates to an *object*. Speaking informally, we'd say that we are storing values in the window.

The window words can be used to store 16-bit USHORT values, 32-bit ULONGs or a 32-bit far pointer, with the following functions:

```
BOOL WinSetWindowUShort(HWND hwnd, SHORT index, USHORT us);
BOOL WinSetWindowULong(HWND hwnd, SHORT index, ULONG ul);
BOOL WinSetWindowPtr(HWND hwnd, SHORT index, PVOID p);
```

In each case, *index* is the zero-based byte offset of the desired data item. For example, if there are three pointers stored in the window words, then the offset of the third pointer would be eight. Values can be retrieved with the following functions:

```
USHORT WinQueryWindowUShort(HWND hwnd, SHORT index);
ULONG WinQueryWindowULong(HWND hwnd, SHORT index);
PVOID WinQueryWindowPtr(HWND hwnd, SHORT index);
```

Now we have a way of storing values which are specific to a window, rather than the class. The most common technique used is to create a data structure which corresponds to the window, and then store a pointer to that structure in the window words.

Parents and Children

An earlier chapter showed a simple sample program which prints "Hello World!" in a single window. Let's expand that example to include two child windows which each contain different messages:

The major structural difference is that we will now need two winprocs: one for the main client window area, as before, plus a new one for the child windows. Notice, *one* winproc for *two* child windows, even though they will contain two different messages.

Other things to look for will include the amended WinRegisterClass() function call for the child window class, with the last parameter being the number of bytes of window words to be reserved.

```
#define INCL_WIN    /* This includes the window manager function
prototypes, */
             /* constants, etc. */
#include <os2.h>    /* This is the main OS/2 definitions include file
*/
```

Now we declare the window procedures: a main winproc and a winproc for the children:

```
/* Prototype for the applications window procedure */
MRESULT EXPENTRY HelloWndProc(HWND hwnd, USHORT usMessage, MPARAM
mp1, MPARAM mp2);
MRESULT EXPENTRY HelloChildWndProc(HWND hwnd, USHORT usMessage,
MPARAM mp1, MPARAM mp2);
```

Next, more handles for the various parent and child frame and client windows:

```
/* We need handles for both the frame window and the client window */
HWND hwndParentFrame, hwndParentClient;
HWND hwndC1Frame, hwndC2Frame, hwndC1Client, hwndC2Client;
```

And for the strings to be displayed:

```
char *pszParentString = "I'm a main window!";
char *pszC1String = "I'm child number one!";
char *pszC2String = "I'm the second child!";
```

An extra classname will be needed, so we declare a static char for it as well as the main classname:

```
/* Multiple windows can be maintained by one wndproc, so it's a
class, rather
than an object. The class is registered by name with the system */
static char szClassName[] = "Hello";
static char szChildClassName[] = "HelloChild";
```

Our program starts off in exactly the same way as the single window version:

```
void main(int argc, char **argv)
{
    HAB hab;       /* Handle to an 'anchor block' */
    HMQ hmq;       /* Handle to the input message queue */
    QMSG qmsg;     /* Queue message structure to hold the incoming
message */
    ULONG ulFrameFlags;   /* Frame creation flags */

    /* First, register with PM to get services. This returns a handle
to an anchor block */
    hab = WinInitialize(0);
    /* Next, create a message queue */
    hmq = WinCreateMsgQueue(hab, 0);
    /* Now register a window class for the application parent window
*/
    if(!WinRegisterClass(hab,
         (PCH)szClassName,
         (PFNWP)HelloWndProc,    /* Pointer to winproc function */
         CS_SYNCPAINT | CS_SIZEREDRAW,   /* Flags to force redraw
when resized, etc. */
         sizeof(char *)))                 /* Number of bytes in
'window words' */
     DosExit(EXIT_PROCESS, 1);
```

Until we register a class for the child windows. Note the use of the `sizeof` operator; a more complex program might use `sizeof(struct _MyWindowData *)` or `sizeof(PPLANWINDATA)` or some such construction:

```
    /* Now register a window class for the application child windows
*/
    if(!WinRegisterClass(hab,
         (PCH)szChildClassName,
         (PFNWP)HelloChildWndProc,    /* Pointer to winproc function
*/
         /* CS_SYNCPAINT | */ CS_SIZEREDRAW,   /* Flags to force
redraw when resized, etc. */
         sizeof(char *)))                 /* Number of bytes in
'window words' */
     DosExit(EXIT_PROCESS, 1);
```

Again, the code looks the same as the single-windowed version:

```
    /* We want a regular window, but without a menu, accelerator
table and icon */
    ulFrameFlags = FCF_STANDARD & ~FCF_MENU & ~FCF_ACCELTABLE &
~FCF_ICON;

    /* Now go ahead and create the window */
    /* This function call returns the frame window handle and also
sets the client window handle */
    hwndParentFrame = WinCreateStdWindow(HWND_DESKTOP,    /* Parent
window */
             WS_VISIBLE,             /* Window style visible */
             &ulFrameFlags,            /* pointer to frame flags */
             (PCH)szClassName,            /* Registered class name */
             "Hello",                /* Text for title bar */
             WS_VISIBLE,             /* Client window style */
             (HMODULE)NULL,          /* Pointer to resource module */
             0,                      /* Resource ID within module */
             (HWND FAR *)&hwndParentClient);        /* Pointer to
client window handle */

    if(hwndParentFrame == 0) {
     WinAlarm(HWND_DESKTOP, WA_ERROR);
     DosExit(EXIT_PROCESS, 1);
    }
```

Until we come to the creation of the child windows:

```
    hwndC1Frame = WinCreateStdWindow(hwndParentClient,
                        WS_VISIBLE,
                        &ulFrameFlags,
                        (PCH)szChildClassName,
                        "Hello Child 1",
                        WS_VISIBLE,
                        (HMODULE)NULL,
                        0,
                        (HWND FAR *)&hwndC1Client);
        if(hwndC1Frame == 0) {
            WinAlarm(HWND_DESKTOP, WA_ERROR);
            DosExit(EXIT_PROCESS, 1);
        }

    hwndC2Frame = WinCreateStdWindow(hwndParentClient,
                        WS_VISIBLE,
                        &ulFrameFlags,
                        (PCH)szChildClassName,
                        "Hello Child 2",
                        WS_VISIBLE,
                        (HMODULE)NULL,
                        0,
                        (HWND FAR *)&hwndC2Client);
        if(hwndC2Frame == 0) {
            WinAlarm(HWND_DESKTOP, WA_ERROR);
            DosExit(EXIT_PROCESS, 1);
        }
```

An alternative design approach I might have used - and do on more complex applications - would be to put the code for the child window creation into the main window's winproc, in the processing of the `WM_CREATE` message.

Now, we set the window words to point to the appropriate strings:

```
    WinSetWindowPtr(hwndParentClient, QWL_USER, pszParentString);
    WinSetWindowPtr(hwndC1Client, QWL_USER, pszC1String);
    WinSetWindowPtr(hwndC2Client, QWL_USER, pszC2String);
```

It's business as usual from this point, at least until the winproc for the child windows:

```
    /* Now loop around processing events. This is called the 'event
loop' */
    /* WinGetMsg returns FALSE when it gets the WM_QUIT message */
    while(WinGetMsg(hab, &qmsg, 0, 0, 0))  /* Get a message */
        WinDispatchMsg(hab, &qmsg);        /* Dispatch it to the
window */

    /* Lastly, clean up */
    WinDestroyWindow(hwndParentFrame);
    WinDestroyMsgQueue(hmq);
    WinTerminate(hab);
}

/* The client window procedure. Gets called with message components
as parameters */
MRESULT EXPENTRY HelloWndProc(HWND hwnd, USHORT usMessage, MPARAM
mp1, MPARAM mp2)
{
    HPS hps;      /* Handle to a presentation space. This is where a
PM program 'draws' */
    RECTL rc;     /* A rectangle structure, used to store the window
coordinates */

    switch(usMessage) {

     case WM_CREATE:     /* process this message by returning FALSE.
This lets the */
         return (MRESULT)FALSE;  /* system continue creating the
window */
         break;

     case WM_ERASEBACKGROUND:    /* Let the frame window procedure
redraw the background */
         return (MRESULT)TRUE;   /* in SYSCLR_WINDOW (usually white)
*/
         break;
```

The processing of the `WM_PAINT` message has to be modified slightly from the single-window version of the program, in two important respects. First, the address of the string to be printed is no longer a static variable (though it could be in this example: there is only one parent window) and so we must use the `WinQueryWindowPtr()` function call to extract it from the window words.

The second modification is less obvious. When one of the child windows is moved or minimised, thereby revealing a previously hidden part of the parent window, the system will invalidate that part (or parts) of the window and send `WM_PAINT` messages to the window. By default, when we call `WinBeginPaint()`, the micro-cached presentation space returned is the size of just the invalidated region. If we went ahead and called `WinDrawText()` to paint some text, it would appear centred in the invalidated region where the child window used to be, rather than centred in the client window area.

For this simple application, the extra overhead of totally repainting the client window area is neither here nor there, and an extra call to `WinQueryWindowRect()` fixes the problem:

```
     case WM_PAINT:    /* The 'guts' of the application */
         hps = WinBeginPaint(hwnd, 0, NULL);   /* Get a presentation
space */
        WinQueryWindowRect(hwnd, &rc);         /* Get the window
dimensions */
         /* Draw the message, in rectangle rc, coloured CLR_NEUTRAL
(black) on
         CLR_BACKGROUND (white), centered, over-writing the entire
rectangle */
         WinDrawText(hps, -1, WinQueryWindowPtr(hwnd, QWL_USER), &rc,
CLR_NEUTRAL, CLR_BACKGROUND,
              DT_CENTER | DT_VCENTER | DT_ERASERECT);
         WinEndPaint(hps);      /* Release the presentation space */
         break;

     case WM_CLOSE:/* User chose CLOSE on system menu or double-
clicked */
        WinPostMsg(hwnd, WM_QUIT, 0, 0);   /* So send back WM_QUIT,
causing */
         break;             /* WinGetMsg to return FALSE and exit
the event loop */

     default:     /* Let the system handle messages we don't */
         return WinDefWindowProc(hwnd, usMessage, mp1, mp2);
         break;
    }
    return 0L;
}
```

The child window is basically the same, except for changed `WM_CLOSE` processing to destroy just this window, rather than exit the application.

```
/* The child window procedure. Gets called with message components as
parameters */
MRESULT EXPENTRY HelloChildWndProc(HWND hwnd, USHORT usMessage,
MPARAM mp1, MPARAM mp2)
{
    HPS hps;      /* Handle to a presentation space. This is where a
PM program 'draws' */
    RECTL rc;     /* A rectangle structure, used to store the window
coordinates */

    switch(usMessage) {

     case WM_CREATE:     /* process this message by returning FALSE.
This lets the */
         return (MRESULT)FALSE;  /* system continue creating the
window */
         break;

     case WM_ERASEBACKGROUND:    /* Let the frame window procedure
redraw the background */
         return (MRESULT)TRUE;   /* in SYSCLR_WINDOW (usually white)
*/
         break;

     case WM_PAINT:    /* The 'guts' of the application */
         hps = WinBeginPaint(hwnd, 0, &rc);   /* Get a presentation
space */
        WinQueryWindowRect(hwnd, &rc);         /* Get the window
dimensions */
         /* Draw the message, in rectangle rc, coloured CLR_NEUTRAL
(black) on
         CLR_BACKGROUND (white), centered, over-writing the entire
rectangle */
         WinDrawText(hps, -1, WinQueryWindowPtr(hwnd, QWL_USER), &rc,
CLR_NEUTRAL, CLR_BACKGROUND,
               DT_CENTER | DT_VCENTER | DT_ERASERECT);
         WinEndPaint(hps);       /* Release the presentation space */
         break;

     case WM_CLOSE:/* User chose CLOSE on system menu or double-
clicked */
        WinDestroyWindow(WinQueryWindow(hwnd, QW_PARENT));
         break;

     default:     /* Let the system handle messages we don't */
         return WinDefWindowProc(hwnd, usMessage, mp1, mp2);
         break;
    }
    return 0L;
}
```

Using Data Structures

More complex examples typically define data structures to store the data items represented by the window. In some cases, these are simple flat structures or possible arrays of structures; in others, the initial data structure contains the head of a linked list of data items, or possible a more complex structure such as a tree.

Here's a more complex example, taken from a real-world application (a flight planning program). Among the other data structures associated with a window is this one, defined in the application's main header file:

```
typedef struct _DATAVIEW {
    NPDATAFILE npDataFile;
    struct _waypoint *buffer;
    LONG recno;
    BOOL RecordChanged;
    struct field *CurrentField;
    struct _DATAVIEW *npdataviewNext;
} DATAVIEW;
```

Because the memory for this structure is allocated and freed as required, we will almost always refer to it via a pointer, and so a pointer to a dataview structure is typedef'ed:

```
typedef DATAVIEW NEAR *NPDATAVIEW;
```

(The near pointer distinction obviously does not apply in OS/2 2.x). Now, rather than keep typing out the `WinQueryWindowPtr()` function call longhand again and again, we use a macro to extract a near pointer to the `DATAVIEW` structure from the client window, given its handle:

```
#define NPDATAVIEWFROMCLIENT(hwndClient)
((NPDATAVIEW)WinQueryWindowPtr(\
     hwndClient, QWS_USER))
```

The window procedure for one of the windows which uses this data structure looks very familiar (I won't show it all here - it's *much* too large!). The basic structure is to extract the pointer to the data structure right at the beginning of the function, since almost every message to be processed will involve the data in the underlying structure:

```
MRESULT EXPENTRY FplFormWndProc(HWND hwnd, USHORT usMessage, MPARAM
mp1, MPARAM mp2)
{

    HPS hps;
    RECTL rcl;

    int field_id;
    NPDATAFILE pdf;
    NPDATAVIEW pdv;

    pdf = NPDATAFILEFROMCLIENT(hwnd);
    pdv = NPDATAVIEWFROMCLIENT(hwnd);
```

Later in the winproc, the variables are accessed by dereferencing through the appropriate pointer. For example, to skip to the next record in the nav waypoint database:

```
     pdv->recno++;
```

and so on.

Window words are an extremely powerful tool; almost any non-trivial program will require the ability to display multiple windows of a single class and keep their data separate yet associated with the appropriate display window. Even when writing almost trivial programs these days, I've taken to defining the appropriate data structures and setting up pointers to them in a window. Subsequent program enhancements become much easier!

Parent - Child Relationships and Window Ownership

In developing Presentation Manager applications, the internal consistency of the system is of enormous value in simplifying the programmer's task. Remember, in PM, almost everything is a window. What the user conceives of as the application's window is, in fact, multiple windows:

Title Bar
System Menu Icon
Minimize & Maximize Buttons
Application Menu Bar
Scroll Bars
Frame Window (and border)
Client Window

As the user interacts with the application, and new 'windows' appear, they also really are windows:

Dialog windows
Static text (yes, this really is a window!)
Text-entry fields
List boxes
Combo boxes (a text-entry field and list box combined)
Multiple-line entry fields
Pushbuttons
Checkbuttons
Radio buttons

OS/2 2.0 introduced even more sophisticated window types, which add value for both the programmer and the user:

Slider controls
Value set controls
Notebook controls
Container controls

All windows behave in the same way: they are created using `WinCreateWindow()` (or `WinCreateStdWindow()`), and their interaction is managed using messages or functions such as `WinSetWindowPos()`, `WinSetWindowText()`, and so on. This makes Presentation Manager much more powerful, yet easier to work with, than competing GUI's.

The Parent/Child Relationship

The various windows which make up an application - and indeed, make up the entire system as the user currently sees it on his desktop - are linked together in parent/child relationships.

All family trees have to start somewhere, and in the case of Presentation Manager, this is with the desktop itself. The desktop is a window which occupies the full screen, and which remains in the background, with application windows partially obscuring it. On some systems, the desktop is a flat grey; on others, the user has personalized it with a bitmap.

On OS/2 2.X, the desktop is actually replaced with a folder called the Desktop Folder, and you might notice as the system boots up that after the initial device driver messages a plain grey screen appears, with the mouse pointer in the center. This is the desktop window. A few seconds later, a zoom effect can be seen as the Desktop Folder is opened in front of it.

So, OS/2 2.1 has two desktops, but from a PM programmer's standpoint, you can ignore the Desktop Folder; it will keep out of our way and we deal only with the original desktop.

If you want to see the desktop window, press Ctrl-Esc to bring up the window list, then press the Shift key as you double-click on the entry for 'Desktop - Icon View' (Shift double-click on a Window List entry will minimise that window - it should be disabled for the desktop, but as you can see, it is not!). You can restore the desktop folder window by Ctrl-double-clicking on its Window List entry.

Windows have handles and ID's; you can refer to the desktop with the manifest constant handle HWND_DESKTOP. Window ID's are compile-time constants defined by the programmer, usually in a header file. Window handles are more useful, but are run-time values allocated by Presentation Manager. However, given a window's ID, and the handle of its parent, you can get its handle by using:

```
hwndTarget = WinWindowFromID(hwndParent, IDTarget);
```

Usually, it is the parent window that needs to know the handles of its children so it can send messages to them, and of course, within a window procedure, that window handle is hwnd. And of course, the programmer knows the ID, since he allocated it.

You should also check out WinQueryWindow(), which is used to find the window handle of a given window's parent, owner, previous, next and so on.

Generally, application frame windows are top-level children of the desktop. So, when you call `WinCreateWindow()` or `WinCreateStdWindow()` to create an application frame, the first parameter - the parent window handle - will be `HWND_DESKTOP`.

Obviously, any window can have multiple children, which will be called siblings. However, each window can have only one parent.

Affected Attributes

Position

The significance of the parent/child relationship is that a child window is positioned relative to its parent. For example, if the x,y coordinates of a child are 50,100, then the child will be drawn 50 pixels to the right and 100 pixels up from the bottom left-hand corner of its parent. If the parent window is moved, then the child moves along with it. So an application window (a top-level window) will be drawn relative to the bottom left-hand corner of the screen (HWND_DESKTOP).

Z-order

Sibling windows can overlap; for example, multiple application windows are often visible simultaneously. The overlapping gives the appearance of depth to the screen. This is referred to as the Z-order; each window has a Z-order position which controls where it lies in the layers of windows. The parent window is always at the bottom of the Z-order stack; in other words, children always appear in front of their parents.

Clipping

However, a child is clipped to its parent. That is, no part of the child - even though it may be larger than the parent, or positioned so that it extends beyond the edges of the parent - will be drawn outside the parent window. This helps to avoid confusion for a user. Generally, all the windows that relate to an application will appear within that application's top-level window, and if a window appears outside that area, then it must be related to some other application.

Of course, it is possible to create dialog windows and other windows which are not clipped to the application's window; just create the window with HWND_DESKTOP as its parent. But care should be taken to ensure that it is clear from context that the window belongs to your application, just to avoid user confusion.

A window can also be prevented from painting over its children by specifying a window style of WS_CLIPCHILDREN, and from painting over its siblings by specifying a window style of WS_CLIPSIBLINGS. These styles clip the window to its children or siblings.

Destruction

Pretty clearly, a child cannot be redrawn relative to a window that no longer exists. Unlike real life, children of PM windows do not outlive their parents. As a convenience to the programmer, when the system destroys a window, it also destroys all its children.

Window Ownership

Window ownership is a completely separate issue from parentage. Often, the parent of a window will also be its owner - but not always.

Ownership relates particularly to a category of windows called control windows. These are the windows that the user interacts with, such as scrollbars, list boxes, text entry fields and so on. These windows mostly take care of their own behaviour; typing text into a text entry field is dealt with by the entryfield code, for example. However, user activity should often be notified back to the application's main window code, in order to control application-specific behaviour - for example, if you press a scroll bar: the scroll bar will update its own appearance automatically, but then must notify its owner, in order to have the application scroll down a document.

When a significant event occurs to a control window, it will send a WM_CONTROL message to its owner. The mp1 parameter will contain a notification code which specifies the event. Here's a subtle complication of PM programming: some events are considered important enough to have their own messages, while others just fall under WM_COMMAND (menu choices and button selections) or WM_CONTROL (events from other window types).

Another subtle point to bear in mind: frame windows have special default behaviour. When a scroll bar, which is owned by a frame window, sends a WM_CONTROL message to that frame to signal some user-initiated event, the frame window has no way of dealing with message. How does the frame know how to scroll your application's window? So instead of doing anything, it passes the message down to the window which has window ID FID_CLIENT, which is, of course, your application's client window. And it does know how to scroll the user's view of the underlying document - or at least, it will, once you've written that part of the code!

In general, messages sent to a frame window, which don't make sense to a frame, are redirected to the appropriate child window. For example, if you send WM_SETWINDOWPARAMS (or use WinSetWindowText()) to set the text in a frame window, it will simply pass that on to the title bar window, changing its text.

Generally, client windows don't have an owner. There is no point in telling anyone else when something happens to a client window - the entire purpose of client windows is to implement application-specific behaviour. And this is why control windows send WM_CONTROL messages to their owners - control windows are, by design, very general and do not implement application-specific behaviour, so they have to send a message to a window that can. So owner windows are usually client or dialog windows, and control windows are owned.

Dealing with Multiple Windows

Most real-world applications have multiple levels of windows. Even the simplest application which produces what the user sees as only one window, really has two levels: the frame and the client.

More complex applications will have other children, such as dialog windows and they in turn will have children in the form of control windows such as entry fields, MLE's, push-buttons and others. The most sophisticated applications will typically have a top-level application window and one or more document windows, allowing the user to deal with several tasks simultaneously.

In this example program, I show one technique for dealing with multiple windows. This is an extension of an earlier program which displayed text in two child windows - in this case, the problem is one of ensuring that the windows are correctly sized and that menu choices go to the correct child window.

In this example, the parentage tree looks like this:

```
hwndParentFrame
        |
        FID_SYSMENU
        |
        FID_TITLEBAR
        |
        FID_MINMAX
        |
        FID_MENU
        |
        hwndParentClient
                |
                hwndC1Frame (FID_SYSMENU, FID_TITLEBAR, FID_MINMAX)
                |       |
                |       hwndC1Client
                |
                hwndC2Frame (FID_SYSMENU, FID_TITLEBAR, FID_MINMAX)
                        |
                        hwndC2Client
```

Note that each window may be known by its handle at run-time (like hwndParent and hwndClient), while others are not (FID_MENU, FID_TITLEBAR) and the handles for these must be found out using the WinWindowFromID() function call.

Notice in the diagram above, that the application's menu is owned by hwndParentFrame, which obligingly relays menu messages down to hwndParentClient - yet many of the menu choices in an application are intended to affect a child window. How can this be dealt with?

The basic philosophy which I subscribe to is that a window should know only what is necessary to get its job done. In essence, this is the idea of encapsulation. So, in this example, the child windows should process messages that set the foreground and background colors, set the quote, etc. The parent window should only do those things that it has to do (in this case, exit) and every other `WM_COMMAND` (or other appropriate) message should be passed down the window tree to the active child window.

The code that does that (the WM_COMMAND processing in the parent window) looks like this:

```
        case WM_COMMAND:
            switch (SHORT1FROMMP(mp1)) {
                case IDM_EXIT:
                    WinPostMsg(hwnd, WM_QUIT, mp1, mp2);
                    break;
                default:
                    WinSendMsg(hwndActive, WM_COMMAND, mp1, mp2);
                    break;
            }
            break;
```

Notice that the default behaviour, for any unidentified menu choice, is to pass it down to the active child window. This means that other menu choices can be added to them, without rewriting the parent window procedure. However, remember that the menu resource is attached to the parent window, so that the isolation is less complete than one might desire.

How do we know which is the active window? There are two basic techniques. The simplest is to ask the system, using the `WinQueryActiveWindow()` function. This being a demonstration program, I've shown another technique, which is to process the `WM_ACTIVATE` messages received by the child windows, and use them to set a global hwndActive variable. While this works, it uses a global variable, something I prefer to avoid, and it introduces yet another interdependence between parent and child. Generally, I'd use `WinQueryActiveWindow()`.

Yet another problem relates to the initial sizing and placement of the child windows. If the programmer does nothing expilicitly about this, the system will draw them in the lower left corner of their parent, and too big to be useable.

If you create both the parent and children windows in the main() function, then you can query the parent's size and then use `WinSetWindowPos()` to position the children. However, it's my philosophy that the main function should create only the top-level window, and then it, in turn, should create the children. Otherwise, how will you handle the File New menu option?

But if you create a child window from within the `WM_CREATE` processing of its parent, you'll find that you can't size it properly. The reason is that `WM_CREATE` is sent to the window *as it is being created* and before it is visible, so that a `WinQueryWindowRect()` call at this time returns a zero size. Using the RECTL from this to size the children results in them having to size.

One fix is to process the `WM_SIZE` message in the parent wndproc and use it to `WinSetWindowPos()` the children. The trouble with this is that every time you resize the parent, it will rearrange the children, which may not be what the user wants. It's a pity the system does not provide a message which is sent to windows just before they are made visible, in order to initially size any children.

Failing the provision of a system message, you can send your own. In this little application, the last thing the parent's `WM_CREATE` processing does is to `WinPostMsg()` a user-defined message to itself. This message won't get read until the window has been created, at which point it uses this message to initially size the child windows.

There are a number of other interesting techniques in this little program, building upon the window words explained in a previous article. For the novice PM programmer, it can serve as the basis of some further exercises, such as adding menu options to allow creation of more child windows (an array of hwnd's? a linked list?), tiling and cascading of the children and full compliance with the Microsoft MDI (Multiple Document Interface) or IBM CUA '91. Other interesting exercises can include allowing the children to add their own options to the parent menu.

```
/* Program demonstrates use of window words, sending menu messages to
    submenus, use of stringtables
    Version 1.0 for OS/2 2.1
*/
#define INCL_WIN    /* This includes the window manager function
prototypes, */
                    /* constants, etc. */
#include <os2.h>    /* This is the main OS/2 definitions include file
*/
#include "malloc.h"
#include <string.h>

#include "quotes.h"

/* Prototype for the applications window procedure */
MRESULT EXPENTRY QuotesWndProc(HWND hwnd, ULONG usMessage, MPARAM
mp1, MPARAM mp2);
MRESULT EXPENTRY QuotesChildWndProc(HWND hwnd, ULONG usMessage,
MPARAM mp1, MPARAM mp2);

/* We need handles for both the frame window and the client window */
HWND hwndParentFrame, hwndParentClient;
HWND hwndC1Frame, hwndC2Frame, hwndC1Client, hwndC2Client;
HWND hwndActive;

/* Multiple windows can be maintained by one wndproc, so it's a
class, rather
than an object. The class is registered by name with the system */
static char szClassName[] = "Quotes";
static char szChildClassName[] = "QuotesChild";

HAB hab;      /* Handle to an 'anchor block' */

void main(int argc, char **argv)
{
    HMQ hmq;       /* Handle to the input message queue */
    QMSG qmsg;     /* Queue message structure to hold the incoming
message */
    ULONG ulFrameFlags;   /* Frame creation flags */

    /* First, register with PM to get services. This returns a handle
to an anchor block */
    hab = WinInitialize(0);
    /* Next, create a message queue */
    hmq = WinCreateMsgQueue(hab, 0);
    /* Now register a window class for the application parent window
*/
    if(!WinRegisterClass(hab,
         (PCH)szClassName,
         (PFNWP)QuotesWndProc,    /* Pointer to winproc function */
         CS_SYNCPAINT | CS_SIZEREDRAW,   /* Flags to force redraw
when resized, etc. */
         sizeof(char *)))                /* Number of bytes in
'window words' */
     DosExit(EXIT_PROCESS, 1);

    /* Now register a window class for the application child windows
*/
    if(!WinRegisterClass(hab,
         (PCH)szChildClassName,
        (PFNWP)QuotesChildWndProc,    /* Pointer to winproc function
*/
        CS_SYNCPAINT | CS_SIZEREDRAW,   /* Flags to force redraw
when resized, etc. */
        sizeof(PSTRCLR)))               /* Number of bytes in
'window words' */
     DosExit(EXIT_PROCESS, 1);

    /* We want a regular window, but without an accelerator table and
icon */
    ulFrameFlags = FCF_STANDARD & ~FCF_ACCELTABLE & ~FCF_ICON;

    /* Now go ahead and create the window */
    /* This function call returns the frame window handle and also
sets the client window handle */
    hwndParentFrame = WinCreateStdWindow(HWND_DESKTOP,    /* Parent
window */
             WS_VISIBLE,             /* Window style visible */
             &ulFrameFlags,            /* pointer to frame flags */
             (PCH)szClassName,            /* Registered class name */
             "Quotable Quotes",               /* Text for title bar
*/
             WS_VISIBLE,             /* Client window style */
             (HMODULE)NULL,          /* Pointer to resource module */
             ID_quotes,                   /* Resource ID within
module */
             (HWND *)&hwndParentClient);      /* Pointer to client
window handle */

    if(hwndParentFrame == 0) {
     WinAlarm(HWND_DESKTOP, WA_ERROR);
     DosExit(EXIT_PROCESS, 1);
    }

    /* Now loop around processing events. This is called the 'event
loop' */
    /* WinGetMsg returns FALSE when it gets the WM_QUIT message */
    while(WinGetMsg(hab, &qmsg, 0, 0, 0))  /* Get a message */
        WinDispatchMsg(hab, &qmsg);        /* Dispatch it to the
window */

    /* Lastly, clean up */
    WinDestroyWindow(hwndParentFrame);
    WinDestroyMsgQueue(hmq);
    WinTerminate(hab);
}

/* The client window procedure. Gets called with message components
as parameters */
MRESULT EXPENTRY QuotesWndProc(HWND hwnd, ULONG usMessage, MPARAM
mp1, MPARAM mp2)
{
    HPS hps;        /* Handle to a presentation space. This is where
a PM program 'draws' */
    RECTL rc;       /* A rectangle structure, used to store the
window coordinates */
    ULONG ulFrameFlags = FCF_STANDARD & ~FCF_MENU & ~FCF_ICON &
~FCF_ACCELTABLE;   /* Frame creation flags */
    USHORT cxNew, cyNew;

    switch(usMessage) {
        case WM_CREATE:     /* process this message by returning
FALSE.  */

            hwndC1Frame = WinCreateStdWindow(hwnd,
                        WS_VISIBLE,
                        &ulFrameFlags,
                        (PCH)szChildClassName,
                        " Child 1",
                        WS_VISIBLE,
                        (HMODULE)NULL,
                        0,
                        (HWND *)&hwndC1Client);
            if(hwndC1Frame == 0) {
                WinAlarm(HWND_DESKTOP, WA_ERROR);
                DosExit(EXIT_PROCESS, 1);
            }

            hwndC2Frame = WinCreateStdWindow(hwnd,
                        WS_VISIBLE,
                        &ulFrameFlags,
                        (PCH)szChildClassName,
                        " Child 2",
                        WS_VISIBLE,
                        (HMODULE)NULL,
                        0,
                        (HWND *)&hwndC2Client);
            if(hwndC2Frame == 0) {
                WinAlarm(HWND_DESKTOP, WA_ERROR);
                WinAlarm(HWND_DESKTOP, WA_ERROR);
                DosExit(EXIT_PROCESS, 1);
            }


            WinSetWindowPtr(hwnd,0, "This is the background window");

            WinPostMsg(hwnd, UM_INITSIZE, 0L, 0L);

            return (MRESULT)FALSE;
            break;

        case UM_INITSIZE:

                WinQueryWindowRect(hwnd, &rc);
                cxNew = rc.xRight;
                cyNew = rc.yTop;

                WinSetWindowPos(hwndC1Frame,
                        HWND_TOP,
                        cxNew / 16,
                        cyNew / 8,
                        cxNew * 3 / 4,
                        cyNew * 3 / 4,
                        SWP_SIZE | SWP_MOVE);

                WinSetWindowPos(hwndC2Frame,
                        HWND_TOP,
                        cxNew/8,
                        cyNew/16,
                        cxNew * 3 / 4,
                        cyNew * 3 / 4,
                        SWP_SIZE | SWP_MOVE);
                break;

        case WM_COMMAND:
            switch (SHORT1FROMMP(mp1)) {
                case IDM_EXIT:
                    WinPostMsg(hwnd, WM_QUIT, mp1, mp2);
                    break;
                default:
                    WinSendMsg(hwndActive, WM_COMMAND, mp1, mp2);
                    break;
            }
            break;

        case WM_ERASEBACKGROUND:    /* Let the frame window procedure
redraw the background */
            return (MRESULT)TRUE;   /* in SYSCLR_WINDOW (usually
white) */
            break;

        case WM_PAINT:      /* The 'guts' of the application */
            hps = WinBeginPaint(hwnd, 0, NULL);   /* Get a
presentation space */
            WinQueryWindowRect(hwnd, &rc);         /* Get the window
dimensions */
            /* Draw the message, in rectangle rc, coloured
CLR_NEUTRAL (black) on
            CLR_BACKGROUND (white), centered, over-writing the entire
rectangle */
            WinDrawText(hps, -1, WinQueryWindowPtr(hwnd, QWL_USER),
&rc, CLR_NEUTRAL, CLR_BACKGROUND,
                        DT_CENTER | DT_VCENTER | DT_ERASERECT);
            WinEndPaint(hps);        /* Release the presentation
space */
            break;

        case WM_DESTROY:
            WinDestroyWindow(hwndC1Frame);
            WinDestroyWindow(hwndC2Frame);
            return (MRESULT) 0;
            break;

        case WM_CLOSE:  /* User chose CLOSE on system menu or double-
clicked */
            WinPostMsg(hwnd, WM_QUIT, 0, 0);    /* So send back
WM_QUIT, causing */
            break;          /* WinGetMsg to return FALSE and exit
the event loop */

        default:       /* Let the system handle messages we don't */
            return WinDefWindowProc(hwnd, usMessage, mp1, mp2);
            break;
    }
    return 0L;
}

/* The child window procedure. Gets called with message components as
parameters */
MRESULT EXPENTRY QuotesChildWndProc(HWND hwnd, ULONG usMessage,
MPARAM mp1, MPARAM mp2)
{
    HPS hps;      /* Handle to a presentation space. This is where a
PM program 'draws' */
    RECTL rc;     /* A rectangle structure, used to store the window
coordinates */
    PSTRCLR p;


    p = PSCFROMHWND(hwnd);

    switch(usMessage) {

        case WM_CREATE:     /* process this message by returning
FALSE. This lets the */
            p = (PSTRCLR) malloc(sizeof(STRCLR));
            WinSetWindowPtr(hwnd,0,(PVOID)p);
            p->ulFColor = CLR_NEUTRAL;
            p->ulBColor = CLR_BACKGROUND;
            strcpy(p->quote, "A suitable quote goes here");
            return (MRESULT)FALSE;  /* system continue creating the
window */
            break;

        case WM_DESTROY:
            free(p);
            return (MRESULT) 0;

        case WM_ACTIVATE:
            if (SHORT1FROMMP(mp1) == TRUE)
                hwndActive = hwnd;
            break;

        case WM_COMMAND:
            switch(SHORT1FROMMP(mp1)) {
                case IDM_TWAIN:
                    WinLoadString(hab, (HMODULE)0, IDS_TWAIN, MAXSTR,
p->quote);
                    break;
                case IDM_IBSEN:
                    WinLoadString(hab, (HMODULE)0, IDS_IBSEN, MAXSTR,
p->quote);
                    break;
                case IDM_GATES:
                    WinLoadString(hab, (HMODULE)0, IDS_GATES, MAXSTR,
p->quote);
                    break;
                case IDMF_RED: p->ulFColor = CLR_RED; break;
                case IDMF_GREEN: p->ulFColor = CLR_GREEN; break;
                case IDMF_BLUE: p->ulFColor = CLR_BLUE; break;
                case IDMB_RED: p->ulBColor = CLR_RED; break;
                case IDMB_GREEN: p->ulBColor = CLR_GREEN; break;
                case IDMB_BLUE: p->ulBColor = CLR_BLUE; break;
            }
            WinInvalidateRect(hwnd, NULL, 0);
            return (MRESULT)0;
            break;

        case WM_ERASEBACKGROUND:    /*Don't let the frame window
procedure redraw the background */
            WinInvalidateRect(hwnd, NULL, FALSE);
            return (MRESULT)FALSE;   /* in SYSCLR_WINDOW (usually
white) */
```

```
            break;

        case WM_PAINT:      /* The 'guts' of the application */
            hps = WinBeginPaint(hwnd, 0, &rc);   /* Get a
presentation space */
            WinQueryWindowRect(hwnd, &rc);         /* Get the window
dimensions */
            /* Draw the message, in rectangle rc, coloured
CLR_NEUTRAL (black) on
            CLR_BACKGROUND (white), centered, over-writing the entire
rectangle */
            WinDrawText(hps, -1, p->quote, &rc, p->ulFColor, p-
>ulBColor,
                        DT_CENTER | DT_VCENTER | DT_ERASERECT);
            WinEndPaint(hps);        /* Release the presentation
space */
            break;

        case WM_CLOSE:  /* User chose CLOSE on system menu or double-
clicked */
            WinDestroyWindow(WinQueryWindow(hwnd, QW_PARENT));
            break;          /* WinGetMsg to return FALSE and exit
the event loop */

        default:       /* Let the system handle messages we don't */
            return WinDefWindowProc(hwnd, usMessage, mp1, mp2);
            break;
    }
    return 0L;
}

/* quotes.rc */

#include "quotes.h"

MENU ID_quotes PRELOAD
BEGIN
    SUBMENU "~File", IDM_FILE
    BEGIN
     MENUITEM "~New" , IDM_NEW
     MENUITEM "~Open...", IDM_OPEN
     MENUITEM SEPARATOR
     MENUITEM "E~xit", IDM_EXIT
    END
    SUBMENU "~Options", IDM_OPTIONS
    BEGIN
     SUBMENU "~Text", IDM_TEXT
     BEGIN
         MENUITEM "~Twain", IDM_TWAIN
         MENUITEM "~Ibsen", IDM_IBSEN
         MENUITEM "~Gates", IDM_GATES
     END
     SUBMENU "~Color", IDM_COLOR
     BEGIN
         SUBMENU "~Foreground", IDM_FOREGROUND
         BEGIN
          MENUITEM "~Red", IDMF_RED
          MENUITEM "~Green", IDMF_GREEN
          MENUITEM "~Blue", IDMF_BLUE
         END
         SUBMENU "~Background", IDM_BACKGROUND
         BEGIN
          MENUITEM "~Red", IDMB_RED
          MENUITEM "~Green", IDMB_GREEN
          MENUITEM "~Blue", IDMB_BLUE
         END
     END
    END
END

STRINGTABLE
BEGIN
    IDS_TWAIN "Thunder is good, thunder is impressive; but it is
lightning that does the work."
    IDS_IBSEN "I hold that man is in the right who is most closely in
league with the future."
    IDS_GATES "After mid-1990, we will ship PM applications at the
same time or before Windows versions."
END

/* QUOTES.H */

#define MAXSTR    128

#define ID_quotes    0x0100
#define IDM_FILE    0x0101
#define IDM_NEW     0x0102
#define IDM_OPEN    0x0103
#define IDM_EXIT    0x0104
#define IDM_OPTIONS 0x0105
#define IDM_TEXT    0x0106
#define IDM_TWAIN   0x0107
#define IDM_IBSEN   0x0108
#define IDM_GATES   0x0109

#define IDS_TWAIN   IDM_TWAIN + 256
#define IDS_IBSEN   IDM_IBSEN + 256
#define IDS_GATES   IDM_GATES + 256

#define IDM_COLOR   0x0300
#define IDM_FOREGROUND  0x0301
#define IDMF_RED    0x0302
#define IDMF_GREEN  0x0303
#define IDMF_BLUE   0x0304
#define IDM_BACKGROUND 0x0305
#define IDMB_RED    0x0306
#define IDMB_GREEN  0x0307
#define IDMB_BLUE   0x0308

#define UM_INITSIZE WM_USER

typedef struct _STRCLR {
    ULONG ulFColor;
    ULONG ulBColor;
    char quote[MAXSTR];
} STRCLR;

typedef STRCLR *PSTRCLR;


#define PSCFROMHWND(hwnd) ((PSTRCLR)WinQueryWindowPtr(hwnd,0))
```

Control Windows

These are windows which the user uses to control an application. In the same way as the dashboard of a car is covered with controls, so the windows of an application also comprises many controls.

Scrollbars

Scrollbars are very common in applications such as word processors and spreadsheets, which often have to display a document which is larger than their window area, and therefore treat the window as a scrollable viewport on a larger document.

Many scrollbars are created automatically as part of the frame/client combination, using WinCreateStdWindow() with the appropriate frame creation flags - FCF_VERTSCROLL for a vertical scroll bar and FCF_HORZSCROLL for a horizontal one. Alternatively, the application may not display scroll bars until the programmer determines that they are necessary, in which case, the WinCreateWindow() function call will be used to create them.

Buttons

There are several types of buttons in Presentation Manager:

Push Buttons

Radio Buttons

Check Buttons

Entry Fields

Entryfield controls are used for free-format text entry. They support drag-through selection, insertion, deletion, cutting, pasting and destructive backspacing. This relives the programmer of considerable work. However, they have no application-driven editing or masking facilities, and this must be added by the application programmer, typically by subclassing.

Multiple Line Entry Fields

The MLE implements a full text editor 'in a box'. It supports word-wrapping, display of text in any font, clipboard operations, word counting and many other operations.

List Boxes

Used to display a list for user selection, these controls are usually vertical, although horizontal list boxes are possible.

Combo Boxes

A combo box is a single control which coordinates the behaviour of both an entry-field and a listbox.

Static Controls

This is a control which displays some static text or a bitmap. It does not generate any WM_CONTROL messages, but its text can be set with the `WinSetWindowText()` function call.

Notebooks

The notebook control is one of the advanced CUA '91 controls which offers a very considerable amount of functionality. Essentially, the notebook coordinates a number of dialog windows using tabs to 'turn' the pages of a notebook, and is extremely useful for reducing complex dialogs to manageable proportions.

Containers

The container control is the most powerful control in OS/2 2.1. It is seen by users as the 'folder' object in the Workplace Shell, and can display data records in iconic, detailed, list, tree or free-format views. Programming containers could be the subject of a course all by itself.

Value Sets

The value-set control provides an array of icons or text buttons ideal for use as a 'toolbar' on a window. Bear in mind, however, that the value-set 'buttons' are not real buttons in that only the first click on them will register.

The last three controls are beyond the scope of this course; however, they are covered in the Advanced OS/2 Programming course.

Business Forms in OS/2

While there are several books around on OS/2 programming, they tend to concentrate on the creation of dialog windows, graphics and games. Nowhere can you find some example code that shows how to create, for example, a simple data entry screen.

Now, it must be said from the outset that if all you want to do is to create a simple business application that will support data entry,some decision making and some report printing, there are *much* easier ways to do it. Easel, Application Manager, Case:PM and ObjectVision are all tools which make life easier for the application programmer and can create generally-useful PM applications at much lower cost than individually-tailored, made-to-measure and hand-stitched C code.

Nonetheless, there are times when you just have to to some low-level coding because that's the way the rest of the application is written and your data entry routine must integrate seamlessly, for example. Plus, the techniques used are of value in other situations. So, with no further apology, here's some code examples that can be used for forms handling.

The example code which follows is a stripped-down version of a full application which is in production. Many features have been removed to protect the copyright-holder (?) but the code as shown will compile and work. For example, the full version is an MDI-compliant version with multiple windows using window words, which tends to obscure the points we are trying to examine here. It contains the minimum necessary to demonstrate a working model.

The features and techniques demonstrated here include:

- A realistically-complex winproc
- Use of child windows (in this case, entry-field controls)
- Dialog windows
- Use of control windows (in this case, buttons and scroll-bar controls)

This application really centres on the definition of a data structure which contains information about a field on the screen. A linked list of such structures constitutes a form.

```
struct field {
    int id;            /* Field id on screen */
    int row;           /* Screen coordinates */
    int col;
    int prow;           /* Prompt coordinates */
    int pcol;
    SHORT type;        /* SQL Server types */
    int length;        /* Field length */
    char *name;     /* Field name - will double as prompt for simple 
apps */
    unsigned char attrib;   /* Screen attributes, not currently used 
in PM */
    int  colsize;  /* SQL Server variable size */
    void *value;    /* SQL Server value, as SQL Server type */
    char *content;  /* Current record value, as char string */
    char *prompt;
    HWND hwndParent;    /* Handle of parent client window */
    HWND hwndPrompt;    /* Handle of prompt static control */
    HWND hwndField;     /* Handle to entry control */
    struct field *prev; /* Pointers to previous and next fields */
    struct field *next; /* used for cursor movement in form-filling 
*/
} ;
```

The structure contains some fields which are not relevant to the current application, because it is also used by full-screen OS/2 VIO and DOS versions of the forms subsystem. Some datatypes are defined using SQL Server data types, because the forms subsystem is also used in developing SQL Server applications. Notice the use of both forward and backwards pointers to adjacent fields, in order to support both tabbing and back-tabbing between fields.

A static array of field structures is declared just prior to the main routine; this provides the field coordinates and prompts for the actual form used in the application. Also included into the man file is the file FORMS.H, which declares the above field structure and also provides prototypes of the various functions used to manipulate and display the form.

The main routine is relatively straightforward and uninteresting. It creates a frame window with the standard controls plus a vertical scroll bar:

```
    ULONG ctldata = FCF_STANDARD | FCF_VERTSCROLL;

    hwndFrame = WinCreateStdWindow(HWND_DESKTOP,
              WS_VISIBLE | FS_STANDARD,
              &ctldata,
              (PCH)szClassName,
              "Flight Plan Navaid Maintenance",
              0L,
              (HMODULE)0,
              ID_RESOURCE,
              &hwndClient);
```

then proceeds to calculate the number of entries in its default database file:

```
    NumWaypoints = statbuf.st_size / sizeof(struct _waypoint);
```

The vertical scroll bar will be used to scroll through the data file, so the next task is to scale the scroll bar so that it reaches the bottom of its shaft just as the user reaches the last record in the database:

```
    hwndScroll = WinWindowFromID(WinQueryWindow(hwndClient,
QW_PARENT), FID_VERTSCROLL);

    WinSendMsg(hwndScroll, SBM_SETSCROLLBAR,
     MPFROM2SHORT(0,0),
     MPFROM2SHORT(0,NumWaypoints-1));
```

Notice the use of `WinQueryWindow()` to get the handle of the client window's parent (in this case it is the frame) and then `WinWindowFromID()` to get the handle of the standard vertical scroll bar. We then send a message to set the scroll bar scale.

After opening the database, the next task is to read the first record and display it:

```
    FpReadRecord(0L);

    f[0].value = buffer.name;
    f[1].value = buffer.state;
    f[2].value = buffer.abbrev;
    f[3].value = buffer.lat;
    f[4].value = buffer.lng;
    f[5].value = buffer.elev;
    f[6].value = buffer.atis;
    f[7].value = buffer.afis;
    f[8].value = buffer.pal;
    f[9].value = buffer.vor;
    f[10].value = buffer.dmei;
    f[11].value = buffer.dme;
    f[12].value = buffer.ndb;
    f[13].value = buffer.bcst;
    f[14].value = buffer.rwinfo;

    disp_form(f);
    fv2c(f);
    refreshform(f);
    RecordChanged = FALSE;
```

This forms technique distinguishes between the value of a field (be it int, float, char or whatever) and the content, expressed as a character string. So, after initialising the field value pointers to point to the fields of the database buffer, `disp_form()` displays the form and `fv2c()` fills in the contents fields with the string equivalents of the values. Finally `refreshform()` copies the contents into the entry fields on screen. From this point onwards, the main routine is straightforward. Things start to get insteresting again with the winproc.

Keyboard Handling

First, we take care of keyboard handling. We ignore key releases; we are interested only in key depressions. Next, we deal with the Up-arrow, Down-arrow, PgUp and PgDn keys by simply sending them to the scroll bar, which conveniently responds to these messages as well as its own.

```
    case WM_CHAR:
        if(CHARMSG(&usMessage)->fs & KC_KEYUP)
         break;;
        switch(CHARMSG(&usMessage)->vkey) {
         case VK_UP:
         case VK_DOWN:
         case VK_PAGEUP:
         case VK_PAGEDOWN:
             return WinSendMsg(hwndScroll, usMessage, mp1, mp2);
             break;
```

The tab and back-tab keys are dealt with by setting the current_field pointer to the next or previous field and setting the keyboard focus to that field. Finally we return true to indicate that the keystroke has been dealt with.

```
         case VK_TAB:
             if(current_field->next != NULL)
              current_field = current_field->next;
             WinSetFocus(HWND_DESKTOP,current_field->hwndField);
             break;
         case VK_BACKTAB:
             if(current_field->prev != NULL)
              current_field = current_field->prev;
             WinSetFocus(HWND_DESKTOP, current_field->hwndField);
             break;
        }
        return (MRESULT)TRUE;
```

Next, we process the various menu command (`WM_COMMAND`) messages. Most are pretty straightforward, until we get to the Search... dialog box:

```
         case IDM_SEARCH:
             if(RecordChanged) {
              FpAskAndWrite(hwnd);
              /* Dealt with the problem, record now OK */
              RecordChanged = FALSE;
             }
             WinDlgBox(HWND_DESKTOP, hwndFrame, Search, (HMODULE)0,
                   ID_SRCH_DLG, NULL);
             FpReadRecord(recno);
             WinSendMsg(hwndScroll, SBM_SETPOS,
              MPFROMSHORT((USHORT) recno),
              MPFROMSHORT(0));
             fv2c(f);
             refreshform(f);
             RecordChanged = FALSE;
             break;
```

Dialog Window

If the current record has been changed, we invoke the FpAskAndWrite routine which will write the modified record out or abandon it. We then invoke a dialog window: the various parameters to the `WinDlgBox()` call are

`HWND_DESKTOP`	The parent of the dialog window; the window on top of which and within which this window will be drawn
`hwndFrame`	The owner of the dialog window; the window which will receive notification messages from this window

`Search`	The name (address) of the winproc for the dialog window
`(HMODULE)0`	Resource module handle; this dialog is a resource in the .EXE file
`ID_SRCH_DLG`	The ID of the search dialog template
`NULL`	pointer to any data (structure) being passed to the dialog window

The `WinDlgBox()` function does not return until the dialog window has been dismissed with the `WinDismissDlg()` function. It then reads the new record, converts its value to form contents, refreshes the form and sets the `RecordChanged` flag.

The WM_COMMAND:ID_PBABANDON message is sent if the user presses the "Abandon" push-button while appending a record. It destroys the various pushbuttons currently on the screen, and then rereads and redisplays the current record:

```
case ID_PBABANDON:
    WinDestroyWindow(hwndButton1);
    WinDestroyWindow(hwndButton2);
    WinDestroyWindow(hwndButton3);
    FpReadRecord(recno);
    fv2c(f);
    refreshform(f);
    RecordChanged = FALSE;
    return(0L);
    break;
```

The various entry-field controls (i.e. text-entry fields) on the screen will send `WM_CONTROL` messages to their owner (which is, of course, the main client window area). There are really only two cases to consider: `EN_SETFOCUS`, which means the user has tabbed onto, or clicked the mouse on, a new field, and `EN_CHANGE`, which means the user has typed something into an entry-field control. In the former case, we extract the entry-field ID from the message and divide by two to get the logical field ID (each entry field has a matching prompt field, which is a static control), check we haven't gone out of bounds, and then set focus on the appropriate field. For `EN_CHANGE`, we simply flag the record as dirty; it will get written out (or not) by the FplAskAndWrite routine when we move off the field.

```
    case WM_CONTROL:
        switch (SHORT2FROMMP(mp1)) {
         case EN_SETFOCUS:  /* New field got focus */
             field_id = SHORT1FROMMP(mp1) / 2;
             if (field_id < current_field->id)
              while (field_id < current_field->id)
                  current_field = current_field->prev;
             if (field_id > current_field->id)
              while (field_id > current_field->id)
                  current_field = current_field->next;
             WinSetFocus(HWND_DESKTOP, current_field->hwndField);
             break;
         case EN_CHANGE:
             RecordChanged = TRUE;
             break;
         default:
             return WinDefWindowProc(hwnd, usMessage, mp1, mp2);
        }
        break;
```

The WM_PAINT message is processed by repainting the background and then redrawing the form on top of it:

```
    case WM_PAINT:
        hps = WinBeginPaint(hwnd, NULL, &rcl);
        WinFillRect(hps, &rcl, CLR_WHITE);
        WinEndPaint(hps);

        WinQueryWindowRect(hwndClient,&ClientSize);
        SetFormPos(f);

        break;
```

Scroll Bars

Scroll bar controls are scaled with SBM_ messages, as we saw earlier, and they send WM_VSCROLL (or WM_HSCROLL) messages to their owner windows. The second half of the message's mp2 parameter contains the specific scroll-bar data, which is extracted using the SHORT2FROMMP cracker macro. The result can be any of several values:

SB_LINEUP	User clicked in the up-arrow
SB_LINEDOWN	User clicked in the down-arrow
SB_PAGEUP	User clicked in the top part of the scroll-bar shaft
SB_PAGEDOWN	User clicked in the lower part of the scroll-bar shaft
SB_SLIDERTRACK	User is dragging the slider. The application must be able to redraw the window quickly in order to keep up with a barrage of these messages. If it can't, then wait for a
SB_SLIDERPOSITION	User released the slider after dragging it
SB_ENDSCROLL	User released the mouse after pressing an arrow or the shaft. No action necessary

The first thing that we must do in processing a scroll message is check to see if the current record is dirty, and if it is, deal with it:

```
if(RecordChanged) {
 FpAskAndWrite(hwnd);
 /* Dealt with the problem, record now OK */
 RecordChanged = FALSE;
}
```

Next, we crack out the mp2 parameter and work out what the user is doing with the scrollbar. We either scroll by one record, or by one-twentieth of the file length (in other words, twenty clicks gets the user through the entire database):

```
switch (SHORT2FROMMP(mp2)) {
 case SB_LINEDOWN:
     recno++;
     break;
 case SB_LINEUP:
     recno--;
     break;
 case SB_PAGEDOWN:
     recno += NumWaypoints / 20;
     break;
 case SB_PAGEUP:
     recno -= NumWaypoints / 20;
     break;
```

If the message is either `SB_SLIDERPOSITION` or `SB_SLIDERTRACK`, then we crack the position from mp2:

```
 case SB_SLIDERPOSITION:
     recno = SHORT1FROMMP(mp2);
     break;
 case SB_SLIDERTRACK:
     recno = SHORT1FROMMP(mp2);
     break;
 default:
     break;
}
```

Finally, we check to make sure we haven't run off the end of the database, then read the record, set the position of the slider and redisplay the form:

```
if (recno < 0)
 recno = 0;
if (recno > NumWaypoints - 1)
 recno = NumWaypoints - 1;
FpReadRecord(recno);
WinSendMsg(hwndScroll, SBM_SETPOS,
 MPFROMSHORT((USHORT) recno),
 MPFROMSHORT(0));
fv2c(f);
refreshform(f);
current_field = f;
RecordChanged = FALSE;
break;
```

If the user has chosen Append, then the client window will have received `WM_COMMAND:IDM_APPENDD` and subsequently called the `Append()` function. This sets the file pointer to the end of the file, overwrites the buffer with blanks, redisplays the form and appends three pushbuttons at the bottom of the screen using the `WinCreateWindow()` function call:

```
    DosChgFilePtr(infile, 0, FILE_END, &ulNewPtr);
    memset(&buffer, ' ',sizeof(struct _waypoint));
    fv2c(f);
    refreshform(f);
    RecordChanged = FALSE;
    current_field = f;
    WinSetFocus(HWND_DESKTOP,current_field->hwndField);

    hwndButton1 = WinCreateWindow(hwndClient,
     WC_BUTTON,
     "OK",
     WS_VISIBLE | BS_PUSHBUTTON | BS_DEFAULT,
     100,10,
     75,38,
     hwndClient,
     HWND_TOP,
     ID_PBOK,
     NULL,
     NULL);
```

The FpReadRecord() and FpWriteRecord() functions are very conventional. The only PM function they call is WinMsgBox() in case of a problem.

Dialog Window Procedure

We now come to the winproc for the Search... dialog box. This is mostly a conventional window procedure, consisting of a switch statement on the message. The only things a user can do to the dialog box are typing in the text fields (handled by their code) or pressing the buttons, which will send `WM_COMMAND` messages with mp1 containing the button ID. If the user presses OK, then we use the `WinQueryDlgItemText()` to extract the text from either the name or abbreviation entry-fields and call a function to do a binary or linear search on the file.

```
    switch (usMessage) {
     case WM_COMMAND:
         switch(SHORT1FROMMP(mp1)) {
          case ID_SRCH_OK:
              WinQueryDlgItemText(hwnd,ID_SRCH_NAME, 25,target);
              if(strlen(target) > 0) {
               temp = FpLocateByName(target);
               if (temp == -1)
                   dbuginfo(hwnd,"FpLocateByName failed");
               else
                   recno = temp;
              }
              else {
               WinQueryDlgItemText(hwnd, ID_SRCH_ABBREV, 5, target);
               if(strlen(target) > 0) {
                   temp = FpLocateByAbbrev(target);
                   if(temp == -1)
                    dbuginfo(hwnd,"FpLocateByAbbrev failed");
                   else
                    recno = temp;
               }
              }
              WinDismissDlg(hwnd,0);
              break;
```

If the user presses the Cancel button, then we need do nothing - just dismiss the dialog.

```
          case ID_SRCH_CANCEL:
              WinDismissDlg(hwnd,0);
              break;
```

The Help button is a special case and sends a `WM_HELP`, rather than `WM_COMMAND`, message. In any case, the help for this program has not yet been implemented, so we call a stub function, `fnyi()` (function not yet implemented), which will pop up a message box. This is a convenient technique, as PM applications must be coded top-down in order to allow testing.

```
     case WM_HELP:
         fnyi(hwnd);
```

Unlike a conventional window procedure, the default handling for a dialog window is a call to `WinDefDlgProc()`:

```
     default:
         return WinDefDlgProc(hwnd, usMessage, mp1, mp2);
```

The remaining functions in the application source file are miscellaneous routines for locating records, displaying message boxes for "Function Not Yet Implemented" and debugging messages and general housekeeping.

The file FORMS.C contains the various functions required to refresh the form on screen, convert from field values to contents and vice versa, add fields to a form and delete them and generally maintain forms. There is nothing in these functions of particular interest to the OS/2 PM programmer - they function equally well under OS/2 VIO and DOS environments - but they are included here as part of the complete application.

The PMFORMS.C file, on the other hand, is PM-specific, and contains the low-level functions for creating, reading and writing, PM fields which actually consist of static controls for the prompts and entry-field controls for the fields themselves.

Placing a field on the screen is done by the `disp_field()` function. The coordinates used in the form structure are expressed in row,column pairs from 0,0 at top left to 23,79 at bottom left. This function does some simple arithmetic to create a `WC_STATIC` control for the prompt and a `WC_ENTRYFIELD` control for the field proper in the correct positions relative to the top left of the window. The `WC_ENTRYFIELD` window, in particular, must be passed an `ENTRYFDATA` structure filled in with the maximum number of characters it may contain as well as the minimum and maximum selectable number of characters.

```
/**** Display a field f on screen *****/

unsigned int disp_field(struct field *f)
{
    ENTRYFDATA efd;

    WinQueryWindowRect(f->hwndParent, &ClientSize);

    f->hwndPrompt = WinCreateWindow(f->hwndParent,
     WC_STATIC,
     f->prompt,
     WS_VISIBLE | SS_TEXT | DT_RIGHT,
     FIELD_X * (f->pcol),(short) ClientSize.yTop - TOP_MARGIN - (f-
>prow * FIELD_Y),
     FIELD_X * strlen(f->prompt), FIELD_HEIGHT,
     f->hwndParent,
     HWND_TOP,
     2 * f->id + 1,
     (PVOID) NULL,
     (PVOID) NULL);

    efd.cb = 1;
    efd.cchEditLimit = (USHORT)f->colsize;
    efd.ichMinSel = 0;
    efd.ichMaxSel = (USHORT)f->colsize;

    f->hwndField = WinCreateWindow(f->hwndParent,
     WC_ENTRYFIELD,
     f->content,
     WS_VISIBLE | ES_AUTOSCROLL| ES_MARGIN,
     FIELD_X * f->col, (short) ClientSize.yTop - TOP_MARGIN + 10 -
(f->row * FIELD_Y),
     f->length >10 ? (FIELD_X-2) * f->length : FIELD_X * f-
>length,FIELD_HEIGHT - 10,
     f->hwndParent,
     HWND_TOP,
     2 * f->id,
     &efd,
     (PVOID) NULL);

    return 0;
}
```

The `SetFieldPos()` function operates by calling the `WinSetWindowPos()` functions for each of the two controls that represent a field, while the field's contents are written (refreshed) and read by calls to `WinSetWindowText()` and `WinQueryWindowText()` respectively.

No attempt has been made in this simple application to provide validation or formatting of either input or output, but this can be achieved relatively easily, by including a pointer to a validation function in the field structure and subclassing the entry-field control.

No claims are made that this application demonstrates the best way to implement forms-handling in the PM environment, but it does work and demonstrates a number of interesting techniques.

File LAB3.C

```
/************************************************************
Flight Planning Database Maintenance Program
by Les Bell 13/8/91
************************************************************/

#define INCL_WIN

#include <os2.h>
#include <process.h>
#define OS21

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys\types.h>

#include "..\forms\forms.h"
#include "..\forms\pmforms.h"
#include "lab3.h"
#include "search.h"
#include "fplan.h"

/* Forward Definitions */
MRESULT EXPENTRY fpmaintWndProc(HWND hwnd, ULONG ulMessage, MPARAM
mp1, MPARAM mp2);
void Append(void);
void FpReadRecord(long recno);
void FpWriteRecord(long recno);
MRESULT EXPENTRY Search(HWND hwnd, ULONG ulMessage, MPARAM mp1,
MPARAM mp2);
void fnyi(HWND owner);
void dbuginfo(HWND owner, PSZ info);
long FpLocateByName(char *id);
long FpLocateByAbbrev(char *abbrev);
void FpAskAndWrite(HWND);

HAB hab;
HMQ hmq;

CHAR szClassName[] = "LBAFPM";

HWND hwndClient;
RECTL ClientSize;
HWND hwndFrame;
HWND hwndScroll;

HWND hwndSelect;
HWND hwndButton1, hwndButton2, hwndButton3;

HFILE infile;
long recno = 0;
long NumWaypoints;
struct _waypoint buffer;
BOOL RecordChanged = FALSE;
APIRET retcode;

/* On-screen form */
struct field f[] =
   {{0,1, 6,1, 1,FLS,25,"name",    0,25,NULL,NULL,"Name",
NULLHANDLE,NULLHANDLE, NULLHANDLE,NULL,f+1},
    {1,1,32,1,26,FLS, 4,"state", 0, 3,NULL,NULL,"State",
NULLHANDLE,NULLHANDLE, NULLHANDLE,f,f+2},
    {2,1,42,1,35,FLS, 4,"abbrev",0, 4,NULL,NULL,"Abbrev",NULLHANDLE,
NULLHANDLE,NULLHANDLE,f+1,f+3},
    {3,2, 6,2, 1,FLS,10,"lat",    0,10,NULL,NULL,"Lat.",
NULLHANDLE, NULLHANDLE,NULLHANDLE,f+2,f+4},
    {4,2,22,2,16,FLS,10,"lng",    0,10,NULL,NULL,"Long.",
NULLHANDLE,NULLHANDLE, NULLHANDLE,f+3,f+5},
    {5,2,39,2,33,FLS, 4,"elev",    0, 4,NULL,NULL,"Elev.",
NULLHANDLE,NULLHANDLE, NULLHANDLE,f+4,f+6},
    {6,3, 6,3, 1,FLS, 5,"atis",    0, 5,NULL,NULL,"ATIS",
NULLHANDLE,NULLHANDLE, NULLHANDLE,f+5,f+7},
    {7,3,20,3,14,FLS, 5,"afis", 0, 5,NULL,NULL,"AFIS", NULLHANDLE,
NULLHANDLE, NULLHANDLE,f+6,f+8},
    {8,3,35,3,30,FLS, 5, "pal", 0, 5,NULL,NULL,"PAL",  NULLHANDLE,
NULLHANDLE, NULLHANDLE,f+7,f+9},
    {9,4, 6,4, 2,FLS, 5, "vor", 0, 5,NULL,NULL,"VOR", NULLHANDLE,
NULLHANDLE, NULLHANDLE,f+8,f+10},
    {10,4,20,4,14,FLS,5,"dmei", 0,
5,NULL,NULL,"DMEI",NULLHANDLE,NULLHANDLE, NULLHANDLE,f+9,f+11},
    {11,4,35,4,30,FLS,2,"dme",    0,
2,NULL,NULL,"DME",NULLHANDLE,NULLHANDLE, NULLHANDLE,f+10,f+12},
    {12,5,6,5, 2,FLS,3,"ndb",    0,
3,NULL,NULL,"NDB",NULLHANDLE,NULLHANDLE, NULLHANDLE,f+11,f+13},
    {13,5,20,5,14,FLS,4,"bcst",    0,
4,NULL,NULL,"Bcst",NULLHANDLE,NULLHANDLE, NULLHANDLE,f+12,f+14},
    {14,6, 6,6, 2,FLS,30,"rwinfo",0,30,NULL,NULL,"Rwy",
NULLHANDLE,NULLHANDLE, NULLHANDLE,f+13,NULL}};

struct field *current_field = f;

/* Main function */

int main(int argc, char *argv[], char *envp[]) {

    ULONG ctldata = FCF_STANDARD | FCF_VERTSCROLL;

    QMSG qmsg;

    hab = WinInitialize(0);
    hmq = WinCreateMsgQueue(hab,0);

    if(!WinRegisterClass(hab,
        (PCH)szClassName,
        (PFNWP)fpmaintWndProc,
        CS_SIZEREDRAW | CS_SYNCPAINT,
        0))
    return(1);

    hwndFrame = WinCreateStdWindow(HWND_DESKTOP,
            WS_VISIBLE | FS_STANDARD,
            &ctldata,
            (PCH)szClassName,
            "Flight Plan Navaid Maintenance",
            0L,
            (HMODULE)0,
            ID_RESOURCE,
            &hwndClient);
    if(hwndFrame == NULLHANDLE)
        return(1);

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

    DosClose(infile);
    WinDestroyWindow(hwndFrame);
    WinDestroyMsgQueue(hmq);
    WinTerminate(hab);
    return(0);
}

MRESULT EXPENTRY fpmaintWndProc(HWND hwnd, ULONG ulMessage, MPARAM
mp1, MPARAM mp2)
{

    HPS hps;
    RECTL rcl;

    ENTRYFDATA efd;

    USHORT usResult;
    int field_id;
    char msgbuf[64];
    struct field *p;
    ULONG usAction;
    FILESTATUS3 *fs;

    switch(ulMessage) {

    case WM_CHAR:
        if(CHARMSG(&ulMessage)->fs & KC_KEYUP) /* Ignore key releases
*/
            break;;
        switch(CHARMSG(&ulMessage)->vkey) {
            case VK_UP:
            case VK_DOWN:
            case VK_PAGEUP:
            case VK_PAGEDOWN:
                /* Take care of arrow, PgUp, PgDn keys */
                return WinSendMsg(hwndScroll, ulMessage, mp1, mp2);
                break;
            case VK_TAB:
                /* Move to next field */
                if(current_field->next != NULL)
                    current_field = current_field->next;
                WinSetFocus(HWND_DESKTOP,current_field->hwndField);
                break;
            case VK_BACKTAB:
                /* Move to previous field */
                if(current_field->prev != NULL)
                    current_field = current_field->prev;
                WinSetFocus(HWND_DESKTOP, current_field->hwndField);
                break;
            }
            return (MRESULT)TRUE;
            break;

    case WM_COMMAND:
        switch (SHORT1FROMMP(mp1)) {
            case IDM_OPEN: /* Open... dialog call */
                break;

            case IDM_SAVE: /* Save As... dialog call */
                break;

            case IDM_QUIT:
                WinPostMsg(hwnd, WM_QUIT, 0L, 0L);
                break;

            case IDM_APPEND:
                Append();
                break;

            case IDM_SEARCH:
                fnyi(hwnd);
                break;

            case ID_PBABANDON:
                WinDestroyWindow(hwndButton1);
                WinDestroyWindow(hwndButton2);
                WinDestroyWindow(hwndButton3);
                FpReadRecord(recno);
                fv2c(f);
                refreshform(f);
                RecordChanged = FALSE;
                return(0L);
                break;

            case ID_PBOK:
                read_form(f);
                fc2v(f);
                NumWaypoints++;

            default:
                WinDefWindowProc(hwnd,ulMessage, mp1, mp2);
        }
        break;

    case WM_CONTROL:
        switch (SHORT2FROMMP(mp1)) {
            case EN_SETFOCUS:    /* New field got focus */
                field_id = SHORT1FROMMP(mp1) / 2;
                if (field_id < current_field->id)
                while (field_id < current_field->id)
                    current_field = current_field->prev;
                if (field_id > current_field->id)
                while (field_id > current_field->id)
                    current_field = current_field->next;
                WinSetFocus(HWND_DESKTOP, current_field->hwndField);
                break;
            case EN_CHANGE:
                RecordChanged = TRUE;
                break;
            default:
                return WinDefWindowProc(hwnd, ulMessage, mp1, mp2);
        }
        break;

    case WM_CREATE:
        fs = malloc(sizeof(FILESTATUS3));

        retcode = DosOpen("FPLAN.DAT",
            &infile,
            &usAction,
            0,
            FILE_NORMAL,
            FILE_OPEN,
            OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYNONE,
            0);

        retcode = DosQueryFileInfo(infile,FIL_STANDARD,fs,
sizeof(FILESTATUS3));
        NumWaypoints = fs->cbFile / sizeof(struct _waypoint);
        free(fs);
        /* Obtain the scroll bar handle */
        hwndScroll = WinWindowFromID(WinQueryWindow(hwnd, QW_PARENT),
FID_VERTSCROLL);

        /* And set its range */
        WinSendMsg(hwndScroll, SBM_SETSCROLLBAR,
        MPFROM2SHORT(0,0),
        MPFROM2SHORT(0,NumWaypoints-1));

        WinQueryWindowRect(hwndClient,&ClientSize);

        f[0].value = buffer.name;
        f[1].value = buffer.state;
        f[2].value = buffer.abbrev;
        f[3].value = buffer.lat;
        f[4].value = buffer.lng;
        f[5].value = buffer.elev;
        f[6].value = buffer.atis;
        f[7].value = buffer.afis;
        f[8].value = buffer.pal;
        f[9].value = buffer.vor;
        f[10].value = buffer.dmei;
        f[11].value = buffer.dme;
        f[12].value = buffer.ndb;
        f[13].value = buffer.bcst;
        f[14].value = buffer.rwinfo;

        p = f;
        while (p) {
            p->hwndParent = hwnd;
            p = p->next;
        }

        FpReadRecord(0L);

        disp_form(f);
        fv2c(f);
        refreshform(f);
        RecordChanged = FALSE;
        return (MRESULT)FALSE;
        break;

    case WM_CLOSE:
        WinPostMsg(hwnd, WM_QUIT, mp1, mp2);
        break;
    case WM_SIZE:
        WinQueryWindowRect(hwnd, &ClientSize);
        SetFormPos(f);
        return (MRESULT) 0;
        break;

    case WM_PAINT:
        hps = WinBeginPaint(hwnd, 0L, &rcl);
        WinFillRect(hps, &rcl, CLR_WHITE);
        WinEndPaint(hps);
        return (MRESULT) 0;
        break;

    case WM_VSCROLL:
        /* Before scrolling, check for record having been edited */

        if(SHORT2FROMMP(mp2) == SB_ENDSCROLL) break;

        if(RecordChanged) {
            FpAskAndWrite(hwnd);
            /* Dealt with the problem, record now OK */
            RecordChanged = FALSE;
        }
        /* Deal with scroll bar messages (SB_LINEDOWN, SB_LINEUP,
SB_PAGEDOWN */
        /* SB_PAGEUP, SB_SLIDERPOSITION, SB_SLIDERTRACK */
        switch (SHORT2FROMMP(mp2)) {
            case SB_LINEDOWN:
                recno++;
                break;
            case SB_LINEUP:
                recno--;
                break;
            case SB_PAGEDOWN:
                recno += NumWaypoints / 20;
                break;
            case SB_PAGEUP:
                recno -= NumWaypoints / 20;
                break;
            case SB_SLIDERPOSITION:
                recno = SHORT1FROMMP(mp2);
                break;
            case SB_SLIDERTRACK:
                recno = SHORT1FROMMP(mp2);
                break;
            default:
                sprintf(msgbuf,"Unknown usCmd value:
%04x",SHORT2FROMMP(mp2));
                dbuginfo(hwnd,msgbuf);
                break;
        }
        if (recno < 0)
            recno = 0;
        if (recno > NumWaypoints - 1)
            recno = NumWaypoints - 1;
        FpReadRecord(recno);
        WinSendMsg(hwndScroll, SBM_SETPOS,
        MPFROMSHORT((USHORT) recno),
        MPFROMSHORT(0));
        fv2c(f);
        refreshform(f);
        current_field = f;
        RecordChanged = FALSE;
        break;

    default:
        return(WinDefWindowProc(hwnd, ulMessage, mp1, mp2));
        break;
    }
    return 0L;
}

void Append()
{
    ULONG ulNewPtr;

    DosChgFilePtr(infile, 0, FILE_END, &ulNewPtr);
    memset(&buffer, ' ',sizeof(struct _waypoint));
    fv2c(f);
    refreshform(f);
    RecordChanged = FALSE;
    current_field = f;
    WinSetFocus(HWND_DESKTOP,current_field->hwndField);

    hwndButton1 = WinCreateWindow(hwndClient,
        WC_BUTTON,
        "OK",
        WS_VISIBLE | BS_PUSHBUTTON | BS_DEFAULT,
        100,10,
        75,38,
        hwndClient,
        HWND_TOP,
        ID_PBOK,
        NULL,
        NULL);

    hwndButton2 = WinCreateWindow(hwndClient,
        WC_BUTTON,
        "Cancel",
        WS_VISIBLE | BS_PUSHBUTTON,
        200,10,
        75,38,
        hwndClient,
        HWND_TOP,
        ID_PBCANCEL,
        NULL,
        NULL);

    hwndButton3 = WinCreateWindow(hwndClient,
        WC_BUTTON,
        "Abandon",
        WS_VISIBLE | BS_PUSHBUTTON,
        300,10,
        75,38,
        hwndClient,
        HWND_TOP,
        ID_PBABANDON,
        NULL,
        NULL);
}

void FpReadRecord(long recno)
```

```
{
    ULONG ulNewPtr;
    ULONG cbBytesRead;

    DosChgFilePtr(infile, (long)recno * sizeof(struct
_waypoint),FILE_BEGIN,&ulNewPtr);
    retcode = DosRead(infile,&buffer, sizeof(struct _waypoint),
&cbBytesRead);
    if(cbBytesRead != sizeof(struct _waypoint)) {
    WinMessageBox(HWND_DESKTOP,
            hwndClient,
            "Could not read record",
            "File Error",
            IDMB_FILEERROR,
            MB_OK | MB_ICONEXCLAMATION);
    DosExit(EXIT_PROCESS,1);
    }
}

void FpWriteRecord(long recno)
{
    ULONG ulNewPtr;
    ULONG cbBytesWritten;

    DosChgFilePtr(infile, (long)recno * sizeof(struct _waypoint),
FILE_BEGIN, &ulNewPtr);
    retcode = DosWrite(infile, &buffer, sizeof(struct _waypoint),
&cbBytesWritten);
    if(cbBytesWritten != sizeof(struct _waypoint)) {
    WinMessageBox(HWND_DESKTOP,
            hwndClient,
            "Could not write record",
            "File Error",
            IDMB_FILEERROR,
            MB_OK | MB_ICONEXCLAMATION);
    DosExit(EXIT_PROCESS,1);
    }
}


void fnyi(HWND owner)
{
    WinMessageBox(HWND_DESKTOP,
    owner,
    "Feature Not Yet Implemented",
    "FP Maintenance",
    ID_FNYI,
    MB_CANCEL | MB_ICONEXCLAMATION);
}


void dbuginfo(HWND owner, PSZ info)
{
    WinMessageBox(HWND_DESKTOP,
    owner,
    info,
    "Debug Info",
    ID_DBGI,
    MB_OK | MB_ICONHAND);
}
```

```
void FpAskAndWrite(HWND hwnd)
{
    USHORT usResult;

    usResult = WinMessageBox(HWND_DESKTOP,
            hwnd,
            (PSZ)"Save this record?",
            (PSZ)"Record Has Been Changed",
            IDMB_SAVE,
            MB_YESNO | MB_ICONQUESTION);
    if(usResult == MBID_YES) {
    read_form(f);
    fc2v(f);
    /* Write back the record */
    FpWriteRecord(recno);
    }
}
```

File LAB3.RC:

```
/* LAB3.RC - File Maintenance Resources */

#include <os2.h>

#include "lab3.h"
#include "search.h"

POINTER ID_RESOURCE "lab3.ico"

ACCELTABLE ID_RESOURCE
BEGIN
"N",IDM_NEW,CONTROL
END

MENU ID_RESOURCE
BEGIN
    SUBMENU "~File", IDM_FILE
    BEGIN
     MENUITEM "~New", IDM_NEW
     MENUITEM "~Open",IDM_OPEN
     MENUITEM "~Save", IDM_SAVE
     MENUITEM SEPARATOR
     MENUITEM "E~xit", IDM_QUIT
    END
    SUBMENU "~Edit", IDM_EDIT
    BEGIN
     MENUITEM "~Undo",IDM_UNDO
     MENUITEM SEPARATOR
     MENUITEM "~Cut", IDM_CUT
     MENUITEM "C~opy", IDM_COPY
     MENUITEM "~Paste", IDM_PASTE
     MENUITEM SEPARATOR
     MENUITEM "~Append",IDM_APPEND
    END
END
```

File MAINT.DEF:

```
; MAINT definition file
; Created by Les Bell and Associates Pty Ltd, 1991
```

```
;
NAME MAINT     WINDOWAPI

DESCRIPTION 'Flight Planner, Copyright (C) 1991 Les Bell and
Associates Pty Ltd'

STUB    'OS2STUB.EXE'

CODE    MOVEABLE
DATA    MOVEABLE MULTIPLE

HEAPSIZE  16384
STACKSIZE 8192
```

File FORMS.H:

```
#pragma pack(1)

struct field {
    int id;            /* Field id on screen */
    int row;           /* Screen coordinates */
    int col;
    int prow;
    int pcol;
    DBTINYINT type;         /* SQL Server types */
    int length;        /* Field length */
    char *name;     /* Field name - will double as prompt for simple
apps */
    unsigned char attrib;   /* Screen attributes, not currently used
in PM */
    DBINT colsize;   /* SQL Server variable size */
    void *value;    /* SQL Server value, as SQL Server type */
    char *content; /* Current record value, as char string */
    char *prompt;
    HWND hwndParent;
    HWND hwndPrompt;
    HWND hwndField;     /* Handle to entry control */
    struct field *prev; /* Pointers to previous and next fields */
    struct field *next; /*  - used for cursor movement in form-
filling */
} ;

extern  struct field *add_field(struct field *where,struct field *p);
extern   struct field *make_field(char *name,int prow, int pcol, int
row,int col,int length, unsigned char type,
    DBINT colsize, unsigned char attrib, char *content, char
*prompt);
extern   void set_field_char(struct field *which,char *what);
extern   void set_field_far_char(struct field *which,char far
*what);
extern  void set_field_int(struct field *which,int i);
extern  void set_field_hex(struct field *which,int i);
extern   unsigned int disp_form(struct field *h);
extern  void release_field(struct field *where);
extern   void unlink_field(struct field *where);
extern  void release_form(struct field *where);
extern  struct field *append_field(struct field *where,struct field
*p);
extern   void clear_rd(int row, int col);
extern   void set_field_ver_no(struct field *which, unsigned int i);
extern   void set_field_flstr(struct field *which, char *pbuf, DBINT
len);
extern   void v2c(struct field *f); /* field value to content
conversion */
extern   void fv2c(struct field *f); /* Convert _all_ values to
contents */
extern   void refreshform(struct field *f);
extern   void read_form(struct field * f);
extern   void fc2v(struct field *f)  /* Convert all form contents to
values */;
extern   void c2v(struct field *f);
extern   void set_flstr_field(struct field *f, char *dest, DBINT
length);

/* Field types */
#define FLS 0x01
```

File FORMS.C:

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <string.h>
#include <graph.h>
#include <string.h>

#define INCL_WIN
#include <os2.h>

#define DBMSOS2
#include <sqlfront.h>
#include <sqldb.h>
#include <stdlib.h>

#include "forms.h"

extern unsigned int disp_field(struct field *f);

/**************** Functions for creating and managing field list
**************/

/********* Add field p into form behind field where ********/

struct field *add_field(struct field *where, struct field *p)
{
    p->prev = where;
    p->next = where->next;
    where->next = p;
    if(p->next != NULL)
     p->next->prev = p;
    return p;
}

/********* Create field and initialize with name, row, col, length,
attrib and content */

struct field * make_field(char *name, int prow, int pcol, int row,
int col, int length,
    unsigned char type, DBINT colsize, unsigned char attrib,
    char *content, char *prompt)
{
    struct field *new;

    if((new = malloc(sizeof(struct field))) == NULL) return NULL;
    if((new->name = malloc(32)) == NULL) return NULL;
    strcpy(new->name,name);
    new->colsize = colsize;
    if((new->value = (void *)malloc((size_t)colsize)) == NULL) return
NULL;
    new->prow = prow;
    new->pcol = pcol;
    new->row = row;
    new->col = col;
    new->length = length;
    new->type = type;
    new->attrib = attrib;
    new->content = content;
    if((new->prompt = malloc(32)) == NULL) return NULL;
    strcpy(new->prompt,prompt);
    new->next = NULL;
    new->prev = NULL;
    return new;
}

/**** Fill field which content from near char string what *****/

void set_field_char(struct field *which, char *what)
{
    if(which->content == NULL)
     which->content = malloc(which->length);
    strcpy(which->content,what);
}

/**** Fill field which content from far char string what *****/
/**** Used e.g. to access strings in ROM BIOS          *****/

void set_field_far_char(struct field *which, char far *what)
{
    char *t, far *f;
    if(which->content == NULL)
     which->content = malloc(which->length);
    /* t = which->content;
    f = what;
    while(f < what + which->length)
     *t++ = *f++; */
    for(t=which->content,f=what;f<what+which->length;*t++=*f++);

}

/**** Fill field which content to decimal from integer i *****/

void set_field_int(struct field *which, int i)
{
    if(which->content == NULL)
     which->content = malloc(which->length);
    itoa(i,which->content, 10);
}

/**** Fill field which content to hex from integer i *****/

void set_field_hex(struct field *which, int i)
{
    if(which->content == NULL)
     which->content = malloc(which->length);
    itoa(i,which->content, 16);
}

/**** Fill field which content with version number from unsigned int
i *****/

void set_field_ver_no(struct field *which, unsigned int i)
{
    if(which->content == NULL)
     which->content = malloc(which->length);
    itoa((i & 0x00f0) >> 4,which->content,10);
    which->content[1] = '.';
    itoa((i & 0x000f), which->content+2, 10);
    which->content[3] = '\0';
}

/***** Display a form (list of fields) with head h on screen ******/

unsigned int disp_form(struct field *h)
{
    while(h != NULL) {
     disp_field(h);
     h = h->next;
    }
    return 0;
}

/***** Release field memory and unlink it from form list ****/

void release_field(struct field *where)
{
    if(where->content != NULL && where->length != 0)
     free(where->content);
    if(where->name != NULL)
     free(where->name);
    unlink_field(where);
    free(where);
}

void unlink_field(struct field *where)
{
    if(where->prev != NULL)
     where->prev->next = where->next;
    if(where->next != NULL)
     where->next->prev = where->prev;
}

/**** Release form memory *******/

void release_form(struct field *where)
{
    struct field *temp;

    while(where->next != NULL) {
     temp = where->next;
     release_field(where);
     where = temp;
    }
}
```

```
/****** Append a field onto end of form *****/

struct field *append_field(struct field *where, struct field *p)
{
    while(where->next != NULL)
     where = where->next; /* Get to bottom of list */
    return add_field(where,p);
}

void set_field_flstr(struct field *which, char *pbuf, DBINT len)
{
    char *p, *q;

    if(which->content == NULL)
     which->content = malloc(which->length +1);

    q = which->content;
    p = pbuf + len - 1;
    while (*p == ' ' && p >= pbuf)
     p--;
    while(pbuf <= p)
     *q++ = *pbuf++;
    *q = '\0';
}

void v2c(struct field *f) /* field value to content conversion */
{
    switch(f->type) {
     case FLS:
         set_field_flstr(f, f->value,f->colsize);
         break;
     default:
         break;
    }
}

void fv2c(struct field *f) /* Convert _all_ values to contents */
{
    while (f != NULL) {
     v2c(f);
     f = f->next;
    }
}

extern void refreshfield(struct field *f);

void refreshform(struct field *f)
{
    while(f != NULL) {
     refreshfield(f);
     f = f->next;
    }
}

extern void read_field(struct field *f);

void read_form(struct field * f)
{
    while(f != NULL) {
     read_field(f);
     f = f->next;
    }
}

void fc2v(struct field *f)  /* Convert all form contents to values */
{
    while (f != NULL) {
     c2v(f);
     f = f->next;
    }
}

void c2v(struct field *f)
{
    switch(f->type) {
     case FLS:
         set_flstr_field(f, f->value,f->colsize);
         break;
     default:
         break;
    }
}


void set_flstr_field(struct field *f, char *dest, DBINT length)
{
    char *s;

    s = f->content;

    while(*s && length) {
     *dest++ = *s++;
     length--;
    }

    while(length) {
     *dest++ = ' ';
     length--;
    }
}
```

File PMFORMS.C

```
#define INCL_WIN
#include <os2.h>

#include <string.h>

#define DBMSOS2
#include <sqlfront.h>
#include <sqldb.h>

#include "e:\develop\sql\forms.h"
#include "e:\develop\sql\pmforms.h"

RECTL ClientSize;

/**** Field positioning and dimension constants */
#define FIELD_HEIGHT    26
#define FIELD_X 10  /* Field horizontal spacing */
#define FIELD_Y (FIELD_HEIGHT + 4)    /* Field vertical spacing */
```

```
#define TOP_MARGIN  10  /* Margin from top of client window area */

/**** Display a field f on screen *****/

unsigned int disp_field(struct field *f)
{
    ENTRYFDATA efd;

    WinQueryWindowRect(f->hwndParent, &ClientSize);

    f->hwndPrompt = WinCreateWindow(f->hwndParent,
     WC_STATIC,
     f->prompt,
     WS_VISIBLE | SS_TEXT | DT_RIGHT,
     FIELD_X * (f->pcol),(short) ClientSize.yTop - TOP_MARGIN - (f-
>prow * FIELD_Y),
     FIELD_X * strlen(f->prompt), FIELD_HEIGHT,
     f->hwndParent,
     HWND_TOP,
     2 * f->id + 1,
     (PVOID) NULL,
     (PVOID) NULL);

    efd.cb = 1;
    efd.cchEditLimit = (USHORT)f->colsize;
    efd.ichMinSel = 0;
    efd.ichMaxSel = (USHORT)f->colsize;

    f->hwndField = WinCreateWindow(f->hwndParent,
     WC_ENTRYFIELD,
     f->content,
     WS_VISIBLE | ES_AUTOSCROLL| ES_MARGIN,
     FIELD_X * f->col, (short) ClientSize.yTop - TOP_MARGIN + 10 -
(f->row * FIELD_Y),
     f->length >10 ? (FIELD_X-2) * f->length : FIELD_X * f-
>length,FIELD_HEIGHT - 10,
     f->hwndParent,
     HWND_TOP,
     2 * f->id,
     &efd,
     (PVOID) NULL);

    return 0;
}

void SetFieldPos(struct field *f)
{
    WinQueryWindowRect(f->hwndParent, &ClientSize);

    WinSetWindowPos(f->hwndPrompt,
     NULL,
     FIELD_X * f->pcol,(short) ClientSize.yTop - TOP_MARGIN - (f-
>prow * FIELD_Y),
     FIELD_X * strlen(f->prompt), FIELD_HEIGHT,
     SWP_MOVE);

    WinSetWindowPos(f->hwndField,
     NULL,
     FIELD_X * f->col, (short) ClientSize.yTop - TOP_MARGIN + 10 -
(f->row * FIELD_Y),
     FIELD_X * f->length,FIELD_HEIGHT - 10,
     SWP_MOVE);
}

void SetFormPos(struct field *f)
{
    while(f != NULL) {
     SetFieldPos(f);
     f = f->next;
    }
}

void refreshfield(struct field *f)
{
    WinSetWindowText(f->hwndField,f->content);
}

void read_field(struct field *f)
{   /* f->colsize +1 to allow for trailing null */
    WinQueryWindowText(f->hwndField,(short)f->colsize+1,f->content);
}
```

The Control Program API

The OS/2 Control Program (also known as the Base Operating System, or the kernel) provides system services for

- Memory management
- Process management (multitasking)
- Thread management (multithreading)
- File system access (I/O)
- Dynamic linking
- Miscellaneous system services (date/time, etc)

These services are accessed through an API (Application Programming Interface). The API used by MS-DOS and PC DOS, for example, consists of

- placing a function number in the AH processor register, and possibly a sub-function in AL
- putting any parameters in DX or ES:BX
- executing a software interrupt instruction (INT 21H). This enters the DOS kernel which then jumps to the appropriate routine

So, for example, to print a string:

```
STR1:    DB 'Error message$'


     .
     .
     MOV AH, 9
     MOV DX, byte offset STR1
     INT 21H
     .
```

This technique cannot be used in OS/2 (other than in a Virtual DOS Machine), since software interrupts are not supported in protected mode. Therefore, the OS/2 API works by direct call to the target function. This offers several benefits:

- no overhead in the OS as it works out the adress of the target function and then dispatches to it
- directly supported by high level languages. All HLL's can generate a CALL instruction, but cannot generate an INT 21H, other by calling a supplied subroutine (more overhead)
- this technique will work in both real and protected modes (although DOS cannot be CALLed, but this objection can be overcome by a binding layer of code)

The only problem to be overcome is this: what *is* the address of the target function?Right now, on this machine, the DosOpen() API resides at a particular address, say ABCDEFH, but on a different machine with a different configuration or different kernel revision, or on the same machine tomorrow with a different set of device drivers, it will be different. How do we know the address to be used?

The answer is, of course, dynamic linking, which is dealt with in another section of this course. The system is able to 'plug in' the correct addresses of the API's as a program loads, so that by the time it starts executing, all addresses (or at least, most of them - see later) are known.

The Control Program API's follow the OS/2 naming conventions, and are all prefixed Dos-, such as `DosOpen()`, `DosQueryProcAddr()`, `DosExitList()`, and so on. Furthermore, the API is extremely orthogonal (unlike Presentation Manager). All the DOS API's return an error code, of type `APIRET`, which is zero on success or an error code on failure.

Of course, this is often not the desired result of the API, for example, when calling `DosAllocMem()` to allocate a memory object, what the programmer wants is a pointer to the memory object - he only wants to know about error codes in the unlikely event of failure. How then is the pointer returned? In the C language, a function can only return a single value of a single type. A structure could be returned, but this is comparatively inefficient. What we need is to pass a reference to the desired variable, and since C does not directly support call by reference, we instead pass a pointer to the variable, i.e. the address of the variable.

At first glance, this can be somewhat confusing for the newcomer. The basic steps and rules to remember are:

- Before using any API look up its definition in the online help. Because the help does not let you cut and paste just part of the window, write the parameters down on a piece of paper. Notice that in the list of parameters, the last one is not a passed parameter (in the formal argument list of the API) but is the returned value.

- Look up the parameters help, and notice whether a parameter is input, output or input/output. If it is input, you will pass its value. If it is output or input/output, you will pass the address of (a pointer to) a variable.

- Look at the type of any output or input/output parameters. They will typically be `PTYPE param`. However, in your code, you should not declare a variable of type `PTYPE` (pointer to TYPE) and pass this variable to the API. Instead, you should declare a variable of type `TYPE`, and pass the *address* of this variable to the API.

For example, here's the definition of a hypothetical API, which is used to obtain a handle for a leg, given an elephant handle and leg number:

```
HELEPHANT hel;          /* Input */
ULONG uLegNumber;       /* Inout */
PHLEG phLeg;            /* Output */
APIRET rc;              /* Return code */

rc = DosQueryElephantLeg(hel, uLegNumber, phLeg);
```

Here's how you would use this in your code:

```
HELEPHANT hJumbo;
ULONG uLegNumber;
HLEG hLeg;
APIRET rc;

.
.
rc = DosQueryElephantLeg(hJumbo, 1, &hLeg);
if (rc) {
     /* Deal with the error here */
}
/* Now go on to do stuff with the leg hLeg */
```

This can become confusing when pointers are involved, as in the memory management API's. For example, the Control Program Reference documents DosAllocMem thusly:

```
 #define INCL_DOSMEMMGR
 #include <os2.h>

 PPVOID    ppb;   /*  A pointer to a variable that will receive the
base address of the allocated private memory object. */
 ULONG     cb;    /*  Size, in bytes, of the private memory object to
allocate. */
 ULONG     flag;  /*  Allocate attribute and desired access
protection flags. */
 APIRET    ulrc;  /*  Return Code. */

 ulrc = DosAllocMem(ppb, cb, flag);
```

You would not declare a `PPVOID` - in fact your application code would probably require a typed pointer such as `PCUSTOMERRECORD` or some such. You would write this as follows:

```
PCUSTOMERRECORD pc;
APIRET rc;

rc = DosAllocMem((PPVOID)&pc, sizeof(PCUSTOMERRECORD), PAG_READ |
PAG_WRITE | PAG_COMMIT);
if (rc) {
     /* Deal with error here */
}
```

Notice that `&pc` is of type `PCUSTOMERRECORD *` or `PPCUSTOMERRECORD`, and this must be cast to a `PPVOID` to keep the compiler quiet at higher warning levels.

In fact, this whole area even confused whoever wrote the Control Program Reference, because if you check the documentation for event sempahores, you'll discover that it refers to a PPHEV, which is a pointer to a pointer to a handle for an event semaphore, whereas actually, what is required is just a pointer to a handle for an event semaphore, or PHEV. One can forgive the author under the circumstances, I think!

Notice that I have always shown an if statement to check the value of the return code from the API. It's good practice to always check return codes, and you should always do this in your code. Throughout this course, I may not show error checking in code fragments and in some of the lab exercises as it tends to obscure the underlying structure, but in real-world programs error checking is mandatory, both during development and at run-time in production systems.

Calling Conventions

OS/2 1.x uses the _Pascal calling convention for its API's. This convention is distinguished by the following characteristics

- Parameters are pushed onto the stack from left to right
- The calling function must rebalance the stack upon resuming execution after the call.

Of course, OS/2 1.x API's will pass 16-bit parameters on the stack, and the pointers passed must be 32-bit (16:16) FAR pointers.

This means that a line of code like

```
    DosBeep(440,1000);
```

will compile to

```
    PUSH 440
    PUSH 1000
    CALL DOSBEEP
    POP BX
    POP BX
```

By contrast, OS/2 2.0 and later passes parameters using the _System calling convention, which is similar to the C calling convention:

- Parameters are pushed onto the stack from right to left
- The called function rebalances the stack upon returning

and so the example above would compile to

```
    PUSH 1000
    PUSH 440
    CALL DosBeep
```

The _Pascal calling convention is faster, which was important in the days of 8 MHz 80286 processors, but the _System calling convention allows for variable numbers of arguments, since we know that the first argument is at the bottom of the stack frame, immediately above the return address. Notice also that the CALL instruction in the 16-bit code is a FAR CALL to another segment, whereas for 32-bit OS/2 2.x/Warp code it is a NEAR CALL. These distinctions become significant when a 32-bit version of OS/2 is running a 16-bit application or subsystem, and vital when writing mixed-model code.

In both cases, the API will return an error code in AX or EAX - zero indicating success - and this becomes the returned value of the function.

The Family API

During the development of OS/2 1.0, Microsoft made a design decision that the API's supported by OS/2 should be a superset of those in DOS. This made possible, or at least easier, two things:

- support for the DOS Compatibility Box, which allows DOS applications to run under OS/2 (and later, OS/2 2.0's MVDM's)

- easy porting of DOS source code to OS/2, so that quickly, thousands of applications would be available

From there, it was a short step to identify a superset of DOS functions and subset of the OS/2 API which could be supported in both environments, together with some memory management rules that would make it possible for a single executable to load and run in both real mode DOS and protected mode OS/2 environments. This API set is referred to as the Family API, and the resultant applications are known as FAPI applications, or bound executables.

The FAPI includes obvious DOS functions such as open/close/read/write files, get/set date/time and so on, but also includes OS/2 functions such as DosBeep() which beeps the PC speaker.

Translation of the OS/2 version of a call such as `DosOpen()` to the corresponding DOS INT 21H function is done by a layer of code known as *bindings.* This library also contains code to implement extensions such as `DosBeep()`, by directly driving the hardware.

The application is written as straight OS/2 code, taking care only to use family API calls, and then compiled and linked in the normal way. However, after linking, an extra bind step is performed, which appends the bindings and a short stub loader onto the end of the .EXE file and also modifies its header.

A 16-bit OS/2 .EXE file actually has two headers. The first is a conventional DOS .EXE file header, but its segment table and entry point cause it to load and run, not the actual program, but a short stub which follows it. Any program can be used here, but the default stub is simply a program which prints "This program cannot be run in real mode" or "This program requires OS/2 Presentation Manager" or similar. The second header is the proper OS/2 header - the OS/2 loader skips over the DOS header and reads this header to load and dynamic link the OS/2 code normally.

In the case of a FAPI application, the DOS header causes the stub loader to load and run, and this in turn loads the application segments, replacing all unresolved references to OS/2 API's with calls to routines in the bindings. Once this is done, the application starts and runs as a conventional DOS program.

Notice the limitations of this technique:

- Since DOS only supports 16-bit code, this can only be done with 16-bit applications
- Only supported by 16-bit compilers, e.g. MS C 6.00 and Watcom C - not VAC++
- Restricted to the Family API - no multithreading, IPC, PM, etc.

However, by querying at run-time which mode it is running in, and then branching appropriately, a FAPI application can contain code which specifically calls BIOS functions in real mode or Vio/Kbd/Mou API's in protected (OS/2) mode, and which can call OS/2-specific API's in that environment.

The astute reader will by now have realised that many of the OS/2 command-line commands must be written this way, since they can execute at both an OS/2 command prompt and a DOS command prompt. And indeed XCOPY, REPLACE, SORT, FIND, FORMAT and other commands are 16-bit code. CHKDSK.EXE is a good example - it provides all of the DOS functionality for checking FAT drives and reporting memory use, but when running in an OS/2 session it does not report memory and even supports HPFS format drives.

Finally, the dual headers on an EXE file can be used to bind together two completely different versions of the same (or hypothetically different) program, one for DOS and one for OS/2. This technique can be used, for example, to bind together a single-threaded character mode DOS version of a program and a 32-bit multithreaded graphical (PM) version for OS/2 - both in the same .EXE file. In fact, because Windows has yet another .EXE header, I have heard tales of people putting DOS, Windows and OS/2 versions of a program in the same file with two bind steps, although I have never tried this personally!

Dynamic Link Libraries

Microsoft and IBM operating systems and applications make extensive use of a type of file called a Dynamic Link Library (DLL). Most users know little about these, other than that applications won't work without them, or crash if the DLL's are in the wrong places. But DLL's are one of the most significant features of both Windows[2] and OS/2, and well-constructed apps which make use of them will integrate well with others. Here's why, together with a brief overview for new developers.

Most people are familiar with the process of compiling and linking an application using a compiled language under DOS. The various source code modules are separately compiled to produce .OBJ files which contain the machine code for their various functions.

However, when an application is constructed this way, all functions must be linked into the resulting single .EXE file. This is the case, whether the function is always required or infrequently used, and can give rise to a situation where the .EXE file is very large - sometimes too large to allow the program to load. This is particularly a problem when the subsystems in the .EXE are rarely invoked simultaneously, such as a help subsystem which will not be used at the same time as a comms subsystem.

By compiling and linking these subsystems as dynamic link libraries, they can be left on disk until required, when they are loaded and linked into the main program. This reduces the size of the main .EXE file, and allows subsystems to be loaded only when required.

While OS/2 does not suffer from limited memory like DOS does, there are other benefits which derive from the use of DLL's. First, when multiple copies of an application are invoked, they can all share the initial module which was loaded earlier. This saves memory and also reduces program load time.

Even more importantly, multiple applications can share a common subsystem. Let's look at a more concrete example. Suppose you are in the business of writing accounting software. Everyone knows that in today's PC market, you can score a major selling point over your competitors if you can persuade other software companies to make their applications interoperate with yours. So, you convince a friend who is writing project management software that he ought to make his application able to import data from your ledger files in order to produce project management reports.

The first way you could solve this problem would be to write out the file formats used by your accounting software, and pass this documentation to your project management programmer friend to write his own routines to access them. If you're feeling particularly generous, you might give him source code for your file-handling routines - though handing out your source code is a commercially dangerous practice. Or you might just give him a .LIB file, containing the compiled file-access routines.

Both parties now link the same (or similar) file-handling routines into their .EXE files, wasting space on the user's disk. But a more serious problem now looms. After both companies have been distributing the software for some months, you discover a bug in your file-handling routines: as the files grow, the B+ trees in your index files aren't being rotated correctly, and in fact, the file structure needs to be changed. You'd also like to update the file structure to add some new features to your software. But if you change anything, it's also going to require

[2] *Much of the discussion which follows does not apply to Windows DLL's, which are very restricted in functionality when compared to those of OS/2. Windows NT provides similar functionality to OS/2.*

your friend to rewrite his project management software and offer his customers an upgrade. You are not going to be popular! Here's the smart way to do things:

Isolate the file-handling routines in a layer of code - in this case, a DLL. Do not allow the higher layers of the application to have any knowledge of the structure of the data files, other than their own record layout. All details of the internal operation of the file handlers - such as whether it uses B+ trees, VSAM, hashing or whatever - to be known to the application. Design a set of file-handling primitives: MyOpenFile(), MyLocateRecord(), MyReadRecord(), MyWriteRecord(), and so on, which will be the only way in which the application gets data.

Now you write the file-handling layer and compile and link its functions into a DLL. The only functions which are exported are the top-level file-handling primitives, which are quite general. Write the application to use these calls, and debug it.

You can now pass the .DLL with accompanying documentation and header files across to your business partner, who writes the project management software to use the same file-handling primitives. You ship your accounting software with the .DLL file to manage the accounts files, while the project management software ships with instructions on how to link to your software through the .DLL if required.

Benefits? Only one set of file-handling routines exist on disk, to be shared by both application programs. But most importantly, should you wish to fix bugs in the file-handling code, you need only supply an updated .DLL and both applications are now fixed! Similarly, should the file format have to be changed, then a file format conversion utility and revised .DLL file will provide improved functionality and both applications will still work!

For many years, smart programmers have expounded the principles of *data hiding* - not allowing higher-layer modules of code to know anything about how lower-level modules represent and manipulate their data internally. This technique leads to good modular design, with fewer bugs and increased re-usability of code. And dynamic link libraries implement the technique beautifully!

More recently, the same concepts have evolved into object-oriented languages and technologies such as C++ and SOM. Stretching the metaphor somewhat, our accounting files represent *objects*, while the code in the DLL represents the *methods* which manipulate the objects. In fact, because a single DLL can manipulate multiple objects for its clients, it could be said to represent a *class*. And now you see what the OS/2 2.0 System Object Model is at least partially about: it provides a standardised mechanism by which new objects can be made known to the operating system. The Workplace Shell and through it, the user, and the methods which manipulate the object can be loaded from their DLL and called by both the parent application and the system.

Types of Dynamic Linking

OS/2 supports two types of dynamic linking. In the first, and easiest to use, the system reads the .EXE file segments as it loads a program, and detects entries which call for dynamic link libraries to be loaded. The DLL's are therefore loaded in turn, and since a .DLL file is structurally similar to a .EXE file, should the DLL reference other DLL's, they will in turn also be loaded, and so the load proceeds, recursively. This is referred to as "load-time" dynamic linking.

This technique has its advantages: it is simple and requires little, if any, modification of source code. However, it has serious drawbacks: all DLL's are loaded at program load-time, consuming memory, and if any of them is missing, the load will fail, with (in OS/2 1.X at least) the cryptic message "The system cannot find the file .". This is appropriate in the case of a missing essential subsystem, but if all that is missing is an optional module or perhaps on-line help, an experienced user could still go ahead and use the program.

Run-time dynamic linking postpones resolution of these subroutine references until the program has loaded and is actually running. In fact, the program can generate the name of a DLL as it runs, then call for the DLL to be loaded and the address of a specific function returned.

This technique has several benefits: first, it can be used to provide a specific pathname for a DLL, allowing loading of a DLL which is not in the LIBPATH (pathnames for DLL's can be stored in .INI files). Second, it allows the application to deal with the situation which arises when a DLL cannot be found, and perhaps continue execution with some options not available. And perhaps most importantly, it allows substitution of DLL's, so that, for example, an application can be developed with a library of DLL's to support import/export of foreign file formats. New DLL's can be added quite simply; the developer need only ensure that the new DLL provides the standard functions (a common .H file can be used to ensure this) and the user need only copy the new DLL to disk and then run a program to update a list of file formats.

Functions of interest:

`DosQueryModuleHandle()`	(DosGetModHandle in OS/2 1.x) Retrieves the handle of a DLL module. Typically used to check whether a DLL has already been loaded.
`DosLoadModule()`	Loads a DLL and returns a module handle, or returns an error code and the name of the DLL which failed to load.
`DosQueryProcAddr()`	(DosGetProcAddr in OS/2 1.x) Takes a module handle and function name, and returns the function address.
`DosFreeModule()`	Takes a module handle and frees the module. The module is only removed from system memory when all client processes have freed it.

Writing and Compiling DLL's

DLL's are supported in both OS/2 1.x and OS/2 2.x, though the two environments use different software development tools. OS/2 1.x developers usually use either IBM C/2 or more commonly Microsoft C 6.00b. Developers for 2.x are almost always using IBM's C Set/2.

Code written to run in a DLL is written in C and assembler and is written in the conventional manner, although there are a few restrictions. First, all functions which are exported must be declared `extern` rather than `static` (extern is the default). `Static` functions cannot be exported.

The source code file for a DLL is compiled in the usual way, but when the linker is run against the .OBJ file, it will run into difficulty as it cannot locate the `main()` function. The linker thinks it is to produce a .EXE file, rather than a DLL.

This is fixed by providing a .DEF (module-definition) file for the linker. A .DEF file usually begins with either a `NAME` statement (for a program .EXE file) or a `LIBRARY` statement (to produce a DLL). Other statements are used to specify, for example, whether code and data segments are preloaded or loaded only when called, stack and heap sizes, and most importantly for a DLL, which functions are exported from the DLL. This last function is achieved with the `EXPORTS` statement. Here's an example:

```
LIBRARY   FORMS
DESCRIPTION 'Forms Handler (C) 1991 Les Bell and Associates Pty Ltd'
PROTMODE
DATA MULTIPLE READWRITE LOADONCALL
CODE LOADONCALL
EXPORTS
     add_field          @1
     make_field         @2
     set_field_char     @3
     set_field_far_char @4
     set_field_int      @5
     set_field_hex      @6
     disp_form          @7
     release_field      @8
     unlink_field       @9
     release_form       @10
     append_field       @11
```

This .DEF file will cause the linker to create a DLL, rather than an .EXE file. The two are very similar; a DOS .EXE file has only one entry point, where the program starts execution with C runtime initialization, whereas an OS/2 DLL has multiple entry points. So do many OS/2 .EXE files: the main() function plus various winprocs. It will also insert a copyright statement into the module, tag the code and data segments with various attributes and define the multiple entry points. Now the linker stops squawking!

When a program which uses a DLL is compiled and linked, some problems arise. The compile will go OK, but the link will fail with "Unresolved external reference" messages as the compiler cannot locate the functions from the DLL. This difficulty can be resolved in several ways.

The first approach is to 'compile' the DLL's DEF file using a tool called the import librarian (`implib.exe`). This will generate a .LIB file with the same name as the DLL but which does not contain any of the actual code of the DLL. Rather it contains some stubs which cause the linker to write the appropriate dynlink records into the resultant .EXE file and postpone linking of the actual code.

An alternative approach is to include an IMPORTS statement in the main module's .DEF file, specifying which modules and functions are loaded from DLL's. The result might look something like this:

```
NAME      MyApp    WINDOWAPI
PROTMODE
IMPORTS
     FORMS.add_field
     FORMS.make_field
     FORMS.set_field_char
     FORMS.set_field_far_char
     FORMS.set_field_int
     FORMS.set_field_hex
     FORMS.disp_form
     FORMS.release_field
     ( . . etc . .)
```

Obviously, at run-time, only the function and module names are known, and there is no run-time protection against passing of bad parameters. This should be dealt with at compile time by creating a .h header file which includes the prototypes of the various functions in the DLL and including it into both the DLL and the main module sources at compile time.

Sharing DLL's

What happens when the power user who has bought both the accounting software and project management program tries to run both together? Now there are two client programs trying to use the file-access routines in a single DLL. Leaving aside the problems of providing file and record locking, how is this dealt with?

DLL code has to be written to be reentrant. Although this sounds difficult, it is actually fairly easy. Remember that in the Intel X86 environment, code segments are either execute-only or execute-and-read; since they cannot be written to, code cannot be self-modifying so that source of difficulty is gone. There is no great problem with clients sharing code segments.

Data is a different matter. We have to arrange that the DLL provides a separate data segment for each client process. Such segments are referred to as *instance data segments*, as they are provided for each instance of the DLL.

By default, a DLL has a single automatic data segment - that is, heap and stack - for all its clients. We can create instance data segments by specifying the MULTIPLE attribute on the DLL's data segments, using the DATA MULTIPLE statement in the .DEF file. Now each client process has its own stack and heap for the DLL.

A DLL can also have a single *global data segment*, which is used for common data. For example, in the case of a multi-client file-handling DLL, there will be some data common to all clients to control file locking and record locking. This will be in a global data segment. Such segments are commonly used by DLL's which manage shared resources such as files, I/O ports and the screen.

Now, in reentrant code, it is a good idea to avoid static and global data as much as possible, as such data is prone to accidental overwrites. In particular, certain sequences of code must be atomic. Examples would include testing, and if clear setting, flags and even simple operations such as incrementing 32-bit long variables on a 16-bit CPU, which requires two machine-code instructions. What can happen if this thread is preempted half-way through incrementing that variable? A second thread might now run and attempt to update the same variable, with disastrous consequences.

Such operations must either be accomplished by calling non-preemptible operating system functions, such as those provided for manipulating semaphores, or by writing critical sections of code, using the OS/2 `DosEnterCritSec()` and `DosExitCritSec()` function calls. While a thread is in a critical section, it cannot be preempted by other threads of the same process, avoiding these problems. Don't crash in a critical section!

Initialization and Termination

When the DLL is loaded, the operating system will call a function to perform any required initialization. By default, an initialization function is provided which will initialize the C run-time data, allowing the DLL code to call C run-time functions. Initialization can be performed on a global or per-instance basis.

This function can be replaced in order to provide customised start-up procedures such as allocating system resources (memory, pipe, queues, etc) or initialising hardware (default screen appearance, for example). However, the initialization must still call the C run-time initialization routine if C functions will be called.

Detailed documentation on writing a replacement initialization routine can be found in the Microsoft and IBM documentation.

Termination code is established by the initialization routine. As each client calls the DLL and it initializes, the routine can call the operating system through the `DosExitList()` function call to register up to 32 routines which will be called when the client process terminates. This allows a DLL to free up resources on the death of a client.

Compiling under MS C 6.00

The segmented memory model used by the 286 processor and OS/2 1.x causes some complications when writing apps which use DLL's. For example, when a thread enters a DLL, the DS register points to the calling application's data segment. In order to access the DLL's data segment, the DS register must be reloaded. In order to achieve this, exported functions (and *only* exported functions) should be declared with the `_loadds` keyword, causing some prolog code to be prepended to the function code.

Of course, this can give rise to a secondary problem. Since I have warned you to avoid static data, you'll probably have written many simple DLL functions to use only automatic (i.e. stack) variables. In that case, your DLL does not have a data segment, and when you link it, the linker will issue segment fixup errors. In that case, get rid of the _loadds keywords or add a single static variable somewhere as a quick fix.

The selection and setup of memory models also gets complicated. Giving the client application access to the DLL data segments (not a good idea - remember my comments about data hiding) will require far pointers, as will the reverse, allowing the DLL access to the client's data segments.

Use the /sfu or /sfw memory model. The w (/Aw) option sets up separate data and stack segments but does not cause reloading of the DS register so that exported functions must be declared with the _loadds keyword. The u (/Au) option will cause DS to be reloaded on every function call, reducing performance.

Declare exported functions with _export _far _pascal, as the functions are, of course, in a different segment from the caller.

Compiling under IBM Visual Age C++

Compiling for the flat memory model of OS/2 2.0 and later using the IBM Visual Age C++ compiler is much simpler.

Compiler options to be used:

/Ge-	Do not generate .EXE file
/Gd-	Statically linked, or
/Gd+	Dynamically linked to C run-time functions.

Visual Age C++ also supports the development of subsystems, which are DLL's that can be shared by multiple processes and which do not use the C run-time environment.

Example Code

Here is an extremely simple example of a DLL. It is a function which is passed three pointers to integers and which returns the sum of the first two integers at the address pointed to by the third pointer.

```
#include <os2.h>

#include "dlldll.h"

void EXPENTRY dlladder(int *a, int *b, int *c)
{
   *c = *a + *b;
}
```

The file dlldll.h simply contains the prototype of the function(s) in the DLL:

```
void EXPENTRY dlladder(int *a, int *b, int *c);
```

The source code file dlldll.c is compiled and linked using the following .DEF file:

```
LIBRARY         DLLDLL INITINSTANCE
DATA            SINGLE SHARED
EXPORTS         DLLADDER
```

The makefile for the dynamic link library shows that as well as compiling and linking the DLL, we also run the Import Librarian to create a DLLDLL.LIB file which will be used to link with the main program:

```
# IBM Developer's Workframe/2 Make File Creation run at 15:48:36 on
07/07/92

# Make File Creation run in directory:
#   C:\DEVELOP\QUICKIES;

.SUFFIXES:

.SUFFIXES: .c

dlldll.dll:  \
  dlldll.OBJ \
  dlldll.def \
  DLLDLL.MAK
   @REM @<<DLLDLL.@0
     dlldll.OBJ
     dlldll.dll


     dlldll.def;
<<
   LINK386.EXE @DLLDLL.@0
  IMPLIB dlldll.LIB dlldll.def

{.}.c.obj:
   ICC.EXE /Ge- /C .\$*.c

!include DLLDLL.DEP
```

The main program which calls this DLL functionis also extremely simple:

```
/* DLL Test Main Program */
#include <os2.h>

#include <stdio.h>

#include "dlldll.h"

main(int argc, char *argv[], char *envp[])
{
    int a = 1,b = 2,c = 0;

    printf("a = %d, b = %d, c = %d\n", a, b, c);
    dlladder(&a, &b, &c);
    printf("a = %d, b = %d, c = %d\n", a, b, c);
    getchar();
}


# IBM Developer's Workframe/2 Make File Creation run at 15:50:02 on
07/07/92

# Make File Creation run in directory:
#   C:\DEVELOP\QUICKIES;

.SUFFIXES:

.SUFFIXES: .c

dllmain.exe:  \
  dlldll.LIB \
  dllmain.OBJ \
  DLLMAIN.MAK
   @REM @<<DLLMAIN.@0
     /CO /ST:8192 /PM:VIO +
     dlldll.LIB +
     dllmain.OBJ
     dllmain.exe


     ;
<<
   LINK386.EXE @DLLMAIN.@0

{.}.c.obj:
   ICC.EXE /C .\$*.c

!include DLLMAIN.DEP
```

Memory Management

Memory Management on the 80286 With OS/2 1.x

OS/2 Version 1.x was designed specifically to run on the Intel 80286 processor, and accordingly its memory management features reflected the segmented architecture of that machine.

The 286 processor generates addresses by combining the values in two 16-bit registers. A segment selector is used to look up the base address of a segment of memory and then an offset is added from one of the general purpose registers.

Each time one of the segment registers is reloaded, there is some delay as the processor reads in the corresponding segment descriptor, and so there is a slight performance penalty associatedwith FAR CALLs and `_far` pointers.

When a program starts running, any C static variables will have been allocated storage in the default heap, while automatic variables are allocated on the stack. In the former case, DS already points to the appropriate segment, while in the later SS is correctly set, and so both are accessed as near.

The C runtime library functions such as `malloc()`, `calloc()`, etc. will allocate memory from the default data segment, so again, in small model, such accesses are near and involve no reloading of the segment registers. However, the default data segment can be at most 64 KB in size and so non-trivial applications usually need additional memory to be allocated by the operating system.

This is done using the `DosAllocSeg()` function call. `DosAllocSeg()`, being a Control Program API, returns a success or failure code, so we must pass a reference to the variable to be set, as a parameter. Furthermore, `DosAllocSeg()` allocates a segment of memory, which is referred to via a selector, and so does not 'return' a pointer as we might wish.

`DosAllocSeg()` takes three parameters: the size of the required segment, the address of a selector to be filled in, and a flags variable which sets segment attributes. After we have obtained a selector, we can convert it into a _far pointer (N.B. this is not the default data segment, so the pointer must be `_far`) by using the MAKEP macro, which combines a selector and offset combination to create a pointer.

```
SEL sel;
PCH pch;
APIRET rc;

rc = DosAllocSeg(size, &sel, SEG_NONSHARED);
pch = MAKEP(sel, 0);
```

Notice that, by definition, the type `PCH` is a `_far` pointer to `char`.

C Library API's - malloc(), etc.

The simplest and most portable technique for memory management is to use the ANSI C function library API's such as `malloc()`, `calloc()` and `free()`. The malloc() function simply attempts to buy memory from the operating system in bulk and then sell it cheaply in smaller quantities. It is simple and it is supported by all C compilers on a huge number of platforms, but it has its problems. Principal amongst these is that the quality of implementation varies widely between compilers, and even between different versions of the same compiler.

`malloc()` will allocate memory from a default heap, with some overhead for a structure called a memory arena which keeps track of the size of this memory block and the offset to the next block.

In the case of 16-bit code, `malloc()` can only allocate memory from the default data segment or heap, the size of which is set by the linker but can never exceed 64 KBytes. Programs which require larger allocations will have to use the underlying operating system functions to allocate segments.

Intel 80386 Memory Management with OS/2 Warp

The segmentation unit of the 386 processor outputs a 32-bit address which is known as a linear address. This is passed to the paging unit, which splits it into three fields. The most significant ten bits provide an index into the page directory, from which the system gets the twenty high-order bits of the page table's physical address. The next ten bits of the linear address are used to index into the page table, and from here the processor obtains the twenty high-order bits of the page frame, i.e. the most significant bits of the physical address, the page number. The remaining 12 bits of the linear address provide the offset of a byte within the page.

In the 386 architecture (and the 486, Pentium and Pentium Pro follow this) pages are fixed in size at 4 KBytes. However, the programmer should not assume this, but should rather use the `DosQuerySysInfo()` API with the `QSV_PAGE_SIZE` parameter to find out for sure.

Each page has attributes such as being readable, writable, committed, and so on, and so OS/2 provides API's for setting and querying these attributes.

OS/2 2.x has been designed to allow for future portability, and so does not follow the segmented model. Nonetheless segmentation is still taking place; but because each segment can be up to 4 GB in size, the program fits entirely within one sgegment and the programmer deals only with offsets, which he can treat as flat addresses within a 4 GB virtual machine.

32-bit OS/2 provides applications with a 512 MB flat address space. The 512 MB limitation comes about because of the need to provide address translation between 32-bit flat-model code and 16-bit segmented-model applications and subsystems (even in Warp, 16-bit code is still used in file system drivers as well as (of course) the DOS/WIN-OS/2 subsystems). The 16-bit code can theoretically have up to 8192 segment descriptors, and if each of these is mapped to a separate 64 KB block of memory, the total address space is 512 MB. Translation between the different models is done by a thunk layer which uses patented translation algorithms.

The operating system kernel actually resides outside the application address space, in fact appearing just below the 4 GB theoretical maximum 32-bit segment limit. However, most of the memory below this - down to 512 MB, in fact - is off-limits to applications, and is only used for system tables.

Within the 512 MB application address space, applications grow upwards from location 00010000H (64KB). The range from 00000000H to 00010000H is an invalid address range, while each application typically starts at 00010000H in its own segment. By contrast, shared memory and dynamic link library memory is allocated from 1FFFFFFFH downwards. Initially, the private address space ends at 03FFFFFFH and this is followed by the 'expansion region'; however the private address space boundary can be moved upwards as required.

Similarly the shared address space area expands downwards; its initial limit is 1BFFFFFFH but it will move downwards into the expansion region as required.

When the system loads an EXE or DLL file, it will allocate memory for the static objects in the file. Once the process starts running, however, the programmer may need to dynamically allocate memory for large structures, linked lists, trees, etc. OS/2 allocates memory objects which can be up to 512 MB in size, using the `DosAllocMem()` API.

The first thing to bear in mind about OS/2 32-bit memory management is that there is a distinct difference between allocation of a memory object and commitment of memory. Allocation of memory causes entries to be made in internal system tables granting an application access to a range of memory. Commitment, by contrast, causes pages of memory to be actually utilised, with consequent reduction in the free memory in the system and possible growth in the system swap file.

Because under the 0:32 memory model, memory objects are not relocatable or resizable in the way that 16:16 segments were, the appropriate strategy for the programmer faced with wide variation in the dynamic memory requirements of his application is to allocate a little more memory than he expects to need, and then only commit it as required. However, caution should be employed - although the memory may never be committed and therefore the swap file may barely grow, the address space itself is a finite resource, and large systems are already running out of address space due to the cumulative effect of several applications which allocate much more memory than they are actually committing or using.

The primary API for memory allocation is `DosAllocMem()`. The syntax is:

```
 #define INCL_DOSMEMMGR
 #include <os2.h>

 PPVOID    ppb;   /*  A pointer to a variable that will receive the
base address of the allocated private memory object. */
 ULONG     cb;    /*  Size, in bytes, of the private memory object to
allocate. */
 ULONG     flags; /*  Allocate attribute and desired access
protection flags. */
 APIRET    rc;    /*  Return Code. */

 rc = DosAllocMem(ppb, cb, flag);
```

Notice that the first parameter is an output parameter, that is, we'd like the API to return a `PVOID`, but since it returns an `APIRET` error code, we pass it the address of a `PVOID` (i.e. a `PPVOID`) which it then fills in.

The primary possibilities for the flags parameter is a combination of the following:

`PAG_READ`	All pages in the object are readable
`PAG_WRITE`	All pages in the object are writable
`PAG_COMMIT`	All pages in the object are committed

Write access actually implies read access, and on the Intel family of processors, PAG_READ and PAG_EXECUTE (pages contain executable code) are also equivalent.

Because the `DosAllocMem()` API maps to the underlying hardware memory management, it always allocates an integral number of pages, which on current Intel processors, are 4 KB in size. Consequently, allocation and committment of a memory object 6 KB in size will result in the allocation of two pages in memory. Accesses within the first 6 KB are treated as expected; however, attempts to access memory beyond the end of the object and within the last page will not trigger an exception as you might expect.

Memory is freed using the `DosFreeMem()` API. The syntax for DosFreeMem() is:

```
 #define INCL_DOSMEMMGR
 #include <os2.h>

 PVOID     pb;    /*  The base virtual address of the private or
shared memory object whose reference is to be freed. */
 APIRET    rc;  /*  Return Code. */

 rc = DosFreeMem(pb);
```

Within a memory object, it is possible to use the allocation flags to commit and decommit, or change the other flags to protect data, on a page-by-page basis. This is done using the `DosQueryMem()` and `DosSetMem()` API's:

```
 #define INCL_DOSMEMMGR
 #include <os2.h>

 PVOID     pb;     /*  The base address of the range of pages to be
queried. */
 PULONG    pcb;    /*  A pointer to the ULONG that contains the size
of the range of pages. */
 PULONG    pFlag;  /*  A pointer to the ULONG in which a set of
attribute flags describing the type of allocation and access
protection for the specified range of pages is returned. */
 APIRET    ulrc;   /*  Return Code. */

 ulrc = DosQueryMem(pb, pcb, pFlag);

 #define INCL_DOSMEMMGR
 #include <os2.h>

 PVOID     pb;    /*  The base address of the range of pages whose
attributes are to be changed. */
 ULONG     cb;    /*  A value specifying the size, in bytes, of the
region whose attributes are to be changed. */
 ULONG     flag;  /*  A set of flags specifying commitment or
decommitment, and desired access protection, for the specified range
of pages. */
 APIRET    ulrc;  /*  Return Code. */

 ulrc = DosSetMem(pb, cb, flag);
```

Note that the last parameter in both cases, the flags, is an output parameter in the query API, and so passed by reference, and an input parameter in the set API, and so passed by value. There are no API's which can change an individual flag, so to make memory read-only, for example, without altering its writable and/or committed status, one must use `DosQueryMem()` to obtain the flags value, AND out the `PAG_WRITE` attribute and then use `DosSetMem()` to write the attributes back again.

In addition, the size of the range of pages to be queried is an input/output parameter. On input, it contains a size (number of bytes) for the page(s) to be queried. The function then determines the state of the first page in the range, and then scans upwards as long as the state of successive pages remains the same. It will stop on encountering a page with different attributes or the end of the object, and set the ULONG pointed to by pcb to contain the number of bytes in the pages for which the attributes are returned.

Suballocating Memory and Heap Management

Because `DosAllocMem()` always rounds up object sizes to a multiple of the page size (at least 4 KB), it is not an efficient API to use when allocating memory for lots of small objects. I have often seen posts in forums like Compuserve's OS2DF1 asking: "Why am I running out of swap space? I'm only allocating 100 bytes at a time, maybe 2 or 3 thousand times". Of course, upon enquiry the poster turns out to be using `DosAllocMem()` so that they are actually allocating (and committing) around 8 or 12 MBytes.

This is where heaps enter the picture. A heap is a pool of memory from which allocation requests are met. Under the covers, this is how malloc() works, for example. A heap is basically a linked list of memory arenas

Here's how it works: there are four phases to heap usage:

- Heap Initialization: Allocation of memory for the heap, and creation of the heap data structures
- Suballocation: acquisition of memory from the heap - repeat as required
- Freeing the suballocations - repeat as required
- Heap cleanup - removal of data structures and freeing of memory

Let's take the example of an application which needs to allocate memory for an unknown number of graphics objects. Each is between 16 bytes and 512 bytes, and not more than 512 KB in total will be required.

Here's the way this code could be written. First we need a function to initialize the heap. In this case, I've made it a little more generic than required (code reuse, don'tcha know):

```
PVOID MemoryInit(ULONG heapsize)
{
    APIRET rc;
    PVOID pheap = NULL;

    rc = DosAllocMem(&pheap, heapsize, PAG_READ | PAG_WRITE |
PAG_COMMIT);
    pmassert(hab, rc == NO_ERROR);

    rc = DosSubSetMem(pheap, DOSSUB_INIT, heapsize);
    pmassert(hab, rc == NO_ERROR);

    return pheap;
}
```

Next, we need an allocation function. Here's one way it could be done:

```
PVOID MemoryAlloc(PVOID pheap, ULONG objsize)
{
    APIRET rc;
    PVOID pobj = NULL;

    rc = DosSubAllocMem(pheap, &pobj, objsize);
    pmassert(hab, rc == NO_ERROR);
    return pobj;
}
```

To free a block from the heap:

```
VOID MemoryFree(PVOID pheap, PVOID pobj, ULONG objsize)
{
    APIRET rc;

    rc = DosSubFreeMem(pheap, pobj, objsize);
}
```

and finally to destroy the heap:

```
VOID MemoryRelease(PVOID pheap)
{
    DosFreeMem(pheap);
}
```

These examples are about as simple as you can get away with. More comprehensive versions would add some checking and debugging facilities.

When using the C library heap management functions (`malloc()`, etc.) remember the `_dump_allocated()` and `_heap_check()` functions. Also compile your program with the /Tm option.

Process Management

In this section, we shall examine OS/2's multitasking facilities and API's.

Preemptive Multitasking

The first thing we need to understand is the nature of the multitasking model used by OS/2. Simpler, and less reliable, systemsoften make use of non-preemptive, or so-called cooperative multitasking. This relies upon each running program doing a little bit of work, then yielding control back to the system, doing a little more work, then yielding, and so on. Each time the application yields and returns control back to the system, it can decide what should run next - the same program or a different one.

This model works acceptably for a totally event-driven system. For example, in a message-passing system, each application will have a loop in which it reads a message and then processes it in some way, and the `GetMessage()` API is the opportunity for the system to wrest control away from the application. But what if processing a task will take a considerable elapsed time? What if, for example, the user chooses File / Open ... and selects a huge document file to be loaded, parsed and displayed? While the application is doing that, it is not calling `GetMessage()` and so the system never regains control - and so no other applications run. The multitasking has just collapsed. This will occur with events that take some time to process (initiating remote sessions, client-server exchanges, etc.), background tasks (printing, etc.) as well as badly written or plain buggy applications which get a message, call a function to process it and never return, thereby crashing the entire system.

It is possible for the application programmer to program his way around this by, for example, periodically peeking the message queue, but heck, you can program your way around the limitations in DOS by the same techniques. The operating system is supposed to be providing multitasking services, right?

By contrast, OS/2 provides preemptive multitasking. In this case, it does not depend upon the program yielding the CPU; instead, the hardware of the machine generates a periodic interrupt which allows the processor to suspend the currently running process, regardless of what it is doing, save its status, work out what should run next and then resume the next task. The programmer need not make any special allowances for running in a multitasking environment, need not divide up a task and periodically yield - in fact, he can just recompile a DOS program with minimal modification and get an OS/2 program which multitasks quite happily.

Sessions

In the OS/2 model, the top-level entities are sessions (also called screen groups in OS/2 1.x). Essentially, each time a full-screen program is launched, OS/2 creates a session for it. A session comprises a virtual video buffer, virtual keyboard and virtual mouse and the one or more processes in the session.

The foreground session, which occupies the screen, has its virtual devices mapped to the physical devices, so that, for example, cursor positioning has a visible effect. Moving the cursor in a background session will have no effect until the session is brought to the foreground, at which time the curor will be correctly updated.

To start a session, the programmer must initialize a `STARTDATA` structure and then call the `DosStartSession()` API. The principal elements of the STARTDATA structure are:

Related - flag to indicate whether the new session has a parent-child relationship with this one

- FgBg - flag to indicate whether the session should start in the foreground
- PgmTitle - pointer to the program title string
- PgmName - the pathname of the program file (may be NULL)
- PgmInputs - command line arguments
- Environment - environment strings
- SessionType - full-screen, windowable, DOS VDM, etc.

The `DosStartSession()` API has many other options, including the ability to start a program from a handle obtained using an API called `WinAddProgram()`, which is no longer supported. Ignore any references to the PgmHandle member in `STARTDATA`. Care needs to be given to the parent-child relationship between sessions, especially when debugging.

The basic `DosStartSession()` call looks like this:

```
STARTDATA    sd;
ULONG        idSession;  /* The ‘returned’ session ID */
PID          pid;        /* The ‘returned’ process ID */

rc = DosStartSession(&sd, &idSession, &pid);
```

Programs

A program is a static entity on disk. For DOS, the definition of a program is easy: it is a .COM or .EXE file. In the OS/2 world, the issue is complicated by the existence of code in dynamic link libraries, but the concept is the same: a program is a static list of code and data stored on disk. It has no life; it does not do anything until the program is invoked, by loading it into memory and running it, at which time it becomes a process.

Processes

When the user issues a command at an OS/2 prompt (or invokes a program from an icon), the operating system loader reads the program's file header and loads the various segments into memory, fixing up address references as it does so. In the process of doing this, it also loads and links in any dynamic link libraries which required, recursively repeating the process with them.

The system also builds up tables of resources allocated to this process, such as memory (allocated during load or later), files, queues and so on. This data is recorded in linked lists, the head of which is in a data structure generically called a process descriptor, and referred to in OS/2 as a Per-Task Data Area.

Starting Child Processes

In the UNIX world, child processes are started by the `fork()/exec()` function call pair. Life under OS/2 is a little simpler and less mystifying. A single function call, `DosExecPgm()`, does it all.

OS/2 has a strong concept of hierarchical inheritance amongst processes. A child process will inherit its parent’s environment (environment strings, file handles, current drive, current directories, etc.) It can change its own environment, and subsequently-created children will inherit the modified environment, but it has no other way of affecting other processes. In addition, a parent process can modify its childrens' thread priorities, but cannot modify priorities in its peers or ancestors.

Another important consequence of this design philosophy is that when a process is terminated or killed by the system, its children will go with it. This is reasonable; often the only reason why the children exist is to perform some service for the parent, and now that it is no longer around, the children have no purpose. Killing the children will avoid deadlocks and free resources (files, semaphores, etc.) for reuse. Programmers should be careful in designing the parent-child relationships for shell programs for this reason. If the shell program is the parent for several critical processes, a trivial bug in the user-interface code of the shell could cause the termination of all the children.

You should use PSTAT|MORE or a graphical process monitor such as PMPS to examine the system process hierarchy. You will notice that there are two copies of `PMSHELL.EXE`. Why?

Everyone assumes that the Workplace Shell is the parent process for user applications, but a little thought will reveal that this cannot be so, otherwise when the Workplace Shell crashes - which it does disturbingly frequently, especially on developers' machines - it would crash all applications the user has invoked from within it. In practice, therefore, the `PROTSHELL=` statement in `CONFIG.SYS` starts a 'master' copy of `PMSHELL.EXE` which maintains the task list and is the true parent. It then reads the `RUNWORKPLACE` environment variable and starts a second copy of `PMSHELL.EXE` which is the user interface. When the user double-clicks on a program reference object to invoke a program, the UI commnicates some parameters to the 'master' shell which then invokes the program. Should the Workplace Shell UI crash, it would only bring down its children, but a look at the process hierarchy will reveal that it has none!

DosExecPgm()

The main API used to kick off programs is DosExecPgm().

system()

The VAC++ compiler continues to support the `system()` library function, which will invoke the command processor (`CMD.EXE`) to run the command passed. For example:

```
int rc;
rc = system("DIR >DIR.TXT");
```

Threads

The OS/2 control program scheduler schedules threads, not processes. A thread is something like a coroutine - a subroutine that executes while its parent, calling, routine continues to execute.

We are all familiar with the concept of subroutines - in fact, most programmers use them with so little conscous thought that we are hardly aware of them!

Figure 1, below, shows subroutine execution, first, from a 'flow of control' viewpoint, and then from an elapsed time perspective. Notice that the main routine is suspended while the subroutine runs.

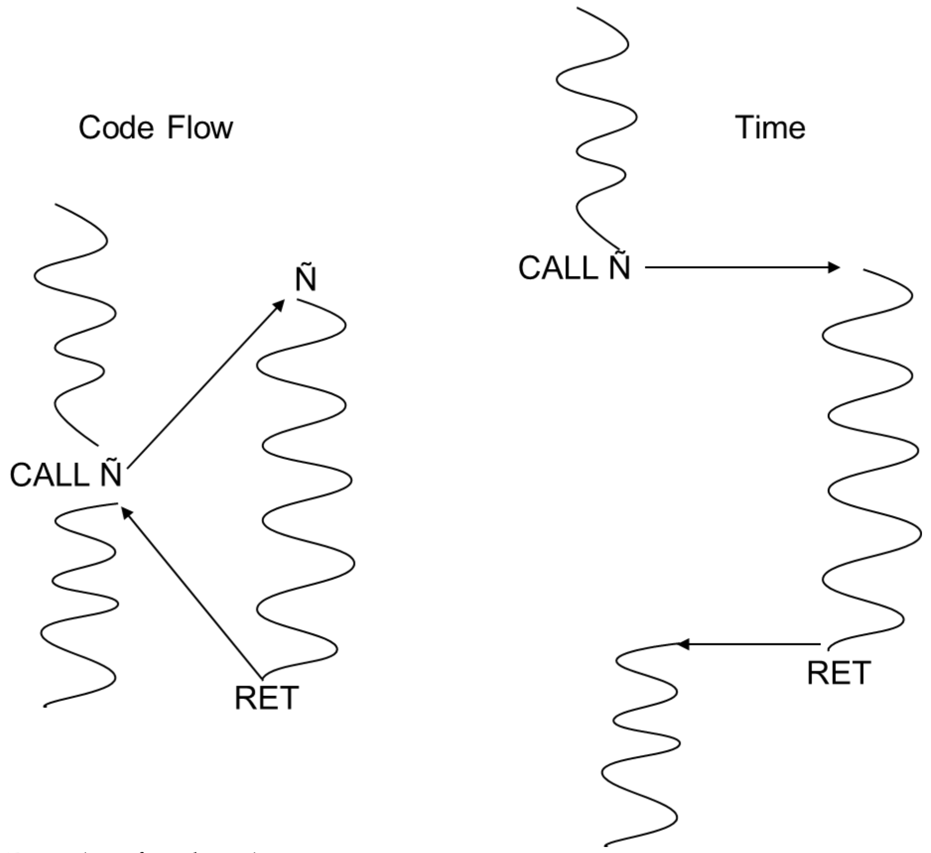


Figure 1. Execution of a subroutine.

But what if we could have the main routine continue while the subroutine runs? This is exactly what OS/2's threads allow you to do. Figure 2 shows a 'subroutine' started by means of the OS/2 `DosBeginThread()` API. Notice that the subroutine starts but the main routine continues to run in parallel with it.

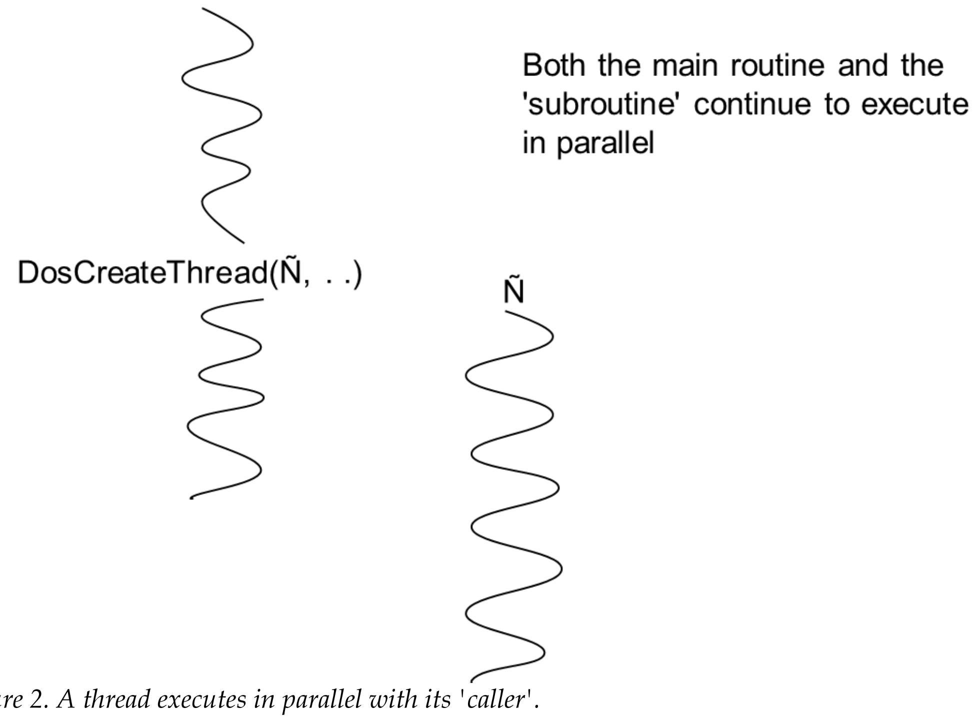

Both the main routine and the 'subroutine' continue to execute in parallel
DosCreateThread(Ñ, . .)
Ñ

Figure 2. A thread executes in parallel with its 'caller'.

Of course, you know that really, at any instant only one instruction is being executed. The processor is time-slicing, switching very rapidly back and forth between the main routine and the subroutine. However, in OS/2 you must never attempt to use this 'fact' in your coding. The reasons are twofold. First, OS/2 is a preemptive multitasking system; this means that you cannot predict when it is going to suspend one thread and resume another. You can make some general statements about the system behaviour, but not specific ones - rather like being unable to chart the random path of an air molecule in a room, while still being able to give useful information about the particle behaviour by stating the air temperature.

Secondly, it's not a 'fact': there are versions of OS/2 which support symmetric multiprocessor hardware, so that the two threads may in fact be executing on two processors so that two instructions do in fact execute at precisely the same instant.

Accordingly, you cannot guarantee (without using system-provided facilities like semaphores) that the two sequences of instructions in two threads will interleave in a particular way. Statements like 'It stands to reason that the first few lines of this thread here will execute before these lines further down the main thread, here' are really a way of asking for trouble.

Threads vs Processes

Why are threads important? Other operating systems like UNIX, seem to get by without them.

The answer is that threads are useful when an application has to do two or more things simultaneously. The classic example is word processing. Way back in 1980, running under the eight-bit CP/M operating system, WordStar version 2.0 was able to print in the background while the user continued to edit a document. Now, it did this by having a loop which polled the keyboard and, if there was no keystroke pending, grabbed another character from the print file and stuffed it out to the printer port. It was done by slightly tricky coding, and certainly not by using any operating system features.

Now, under UNIX, the simultaneous edit and print problem is easily dealt with - just make editing and print formatting separate programs. Those who have used UNIX and its tortuous editors, emacs and vi, together with print formatters like nroff, troff and ditroff will know what I mean. But real-world end users, God Bless Them, think that word processing should be One Program, with menu options for printing. OK, no problem - make the editor program a shell, and it can use `fork()/exec()` or `DosExecPgm()` to invoke a print formatter program 'under the covers'. But look at the overhead here - every time you want to print, the OS has to allocate memory for system structures, load and parse an executable file (and possibly DLL's as well), allocate more memory and so on - all the overhead that goes with process creation.

Threads provide lightweight multitasking. Here's why - and there are a couple of important definitions to bear in mind:

- The process is the OS/2 instance of program execution. It is also the unit of resource ownership.
- A thread is a dispatchable entity. It is the OS/2 unit of execution.

In other words, processes own things. Threads do not. Let me just repeat that:

Processes own things. Threads do not.

What sort of things do processes own? They can own:

- Files
- Semaphores
- Pipes
- Queues
- Windows (though there are some complicating factors here in PM)
- Memory

In short, anything that has a handle or an address. Notice that, although memory allocation may be done by a thread, the memory does not belong to the thread, but to the process. All threads in a process enjoy equal access to the memory and other resources owned by the process. In other words, OS/2 implements inter-process protection, so that processes cannot (by default) screw around with each others' memory, but *there is no inter-thread protection*. Similar considerations apply to files: the `DosOpen()` API may be called by a thread, but the file belongs to the process.

A thread really only 'owns' two things: its priority and its state as represented by the processor registers. The thread has a stack where it pushes automatic variables and return addresses, but it does not 'own' that stack and other threads could overwrite it - for example, with a rogue pointer.

Notice also that the dispatcher works on threads, not processes. In other words, if two processes are running, and one of them comprises a single thread while the other has nine threads, then - all else being equal, which it never is, and assuming all threads are able to run - out of every ten timeslices, the first process would get one while the second process would get nine.

Now, don't get all excited and start to dream of making your applications run amazingly fast by putting a dozen threads to work where your competitors only have one. It just doesn't work that way. In addition, users will notice that while your application may be fast, it is making everything else run slowly - and they'll dump your app in favour of one that has less impact on the rest of the system. Life is full of compromises.

So, coming back to the example of background printing: an OS/2 programmer would put print formatting into a thread rather than a separate process. The print formatting code could already be loaded, ready to run. The app would simply snapshot the data to be printed and then turn the print thread loose to run in the background while the main thread continues editing or whatever.

To summarise: threads are much cheaper than using separate processes, but you trade off some protection. It's no big deal, and even if you sometimes do use a separate process, at least you had the option.

Uses for Threads

What else can we do with threads? All kinds of stuff. Background printing is just the start. How about recalculation in spreadsheets? Rather than have to write some tricky code that checks the keyboard for a keystroke, and if there's nothing there, does a bit more of the recalculation, then comes back to check the keyboard status again, why not just put recalculation in a separate thread?

Now, of course, some complications start to emerge. What if the user edits a cell while a recalculation is in progress? After the value has already been used? There are two approaches here: one is to use a semaphore or some other mechanism to lock out user changes to spreadsheet values while recalculation is in progress. Another is to use some form of inter-thread communication to signal the recalc thread to go back and recalc as far as necessary when a cell is changed. Similar concerns apply when printing.

In fact, thinking about spreadsheets, here are some more ideas: how about allowing macros to run on 'background' threads while the main spreadsheet interface remains active? Of course, now you'd have to provide some mechanism for 'cell locking' so that a user can't update a cell while a macro is using it. Or how about allowing multiple macros to run simultaneously?

Other uses for threads relate to making the system more responsive (generally, threads can't be used to provide straight-out speed improvements. If a calculation takes 100 million instructions, that's what it takes, even if you could dice it up ten ways). Here are some examples:

- Reading records in advance from a mainframe, while letting the user start editing the first

- Keyword indexing a document database while the user reads the first document

- Write-behind caching: letting the user edit the next record while still writing the previous to disk, remote host or whatever

- Filling a list box in the background while the user types into an entry-field (just in case the user needs to pick from the list)

- Painting complex graphics windows in the background while the user interaces with other parts of the application e.g. print format dialogs

- Memory management and garbage collection without holding up the user.

And so on, and so on. There's no real limit to what you can do with threads and a little imagination. And once you've got the basics nailed down, they're not hard to use.

A favourite example of how threads can actually simplify designs comes from the world of comms and networking - an area where OS/2 excels, and now you'll see why.

Take a look at Figure 3. This shows the basic architecture of a terminal emulation program for a single-threaded environment.

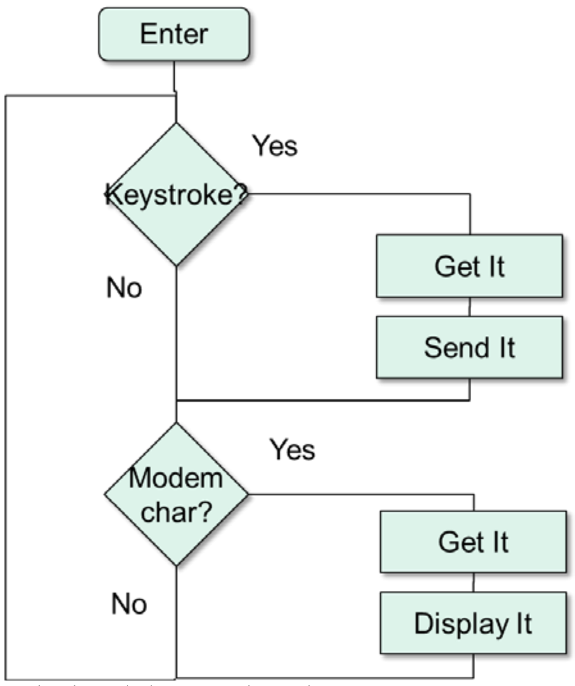


Figure 3. A traditional single-threaded terminal emulation program.

There are several problems with this design. The major one is that the program consists of a single loop with two functions - one for sending keystrokes out the modem and one for receiving and displaying characters. It's not possible to, for example, wait for a keystroke from the user, because if the program stops and waits, it will no longer be monitoring the serial port for incoming characters.

Now think about what happens when the program is idling - nothing being typed at the keyboard, no incoming characters from the modem. What is happening? Right - the program is spinning its wheels, going around and around that main loop but not actually doing anyhing but slurping up CPU cycles and wasting them. That may be OK in a DOS environment, where there's nothing else running anyway, but in a multitasking environment it will degrade the performance of everything else that's running.

There are some other, lesser problems with the design: first, there's too tight coupling between the send and receive sides of the program; secondly, there's no provision for interpretation of escape sequences and adding that will be messy due to the aforementioned coupling; thirdly, there's no provision for buffering keystrokes should the user be able to type faster than they can be sent or should the host be able to send faster than they can be displayed.

Now, here's the multi-threaded approach used in OS/2. Take a look at figure 4. The first thread is the main thread, which will go on to process user commands (from menus, Alt-key options, or whatever). All it has to do is to create a transmit thread and a receive thread.

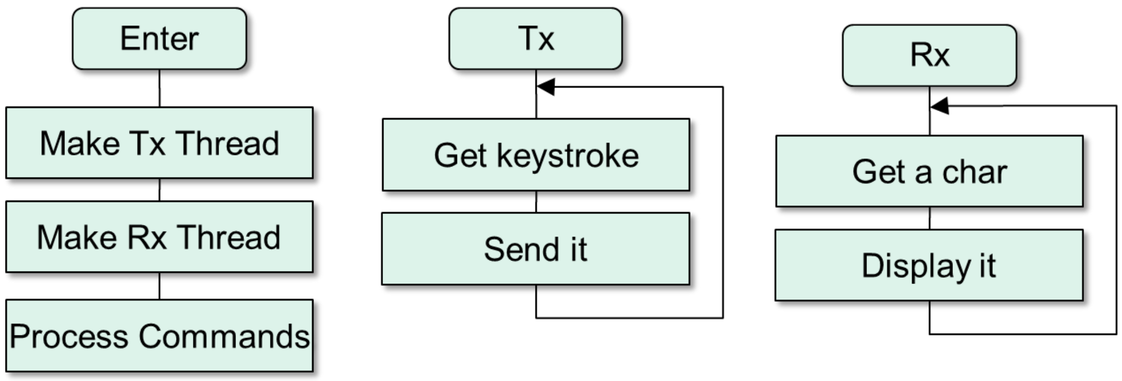


Figure 4. A multithreaded terminal emulator design.

The transmit thread only has to get a keystroke, and then send it. Notice, it will call an OS function to get a keystroke, and wait until it has got it. Only then does it resume execution and send it. Similarly, the receive thread will call the OS to get a character from the serial port, and then display it. Since the two threads operate independently, either can block, waiting for an event, without holding up the other. Notice that now, when the system is idling, no CPU cycles are being used: the main thread is blocked waiting for user commands, the transmit thread is blocked waiting for a keystroke and the receive thread is blocked waiting for an incoming character. Result: minimal load on the system and no degradation of other applications.

In addition, this design is easy to 'open up' and extend, for example, to add double-buffering. Figure 5 shows an emhanced design.

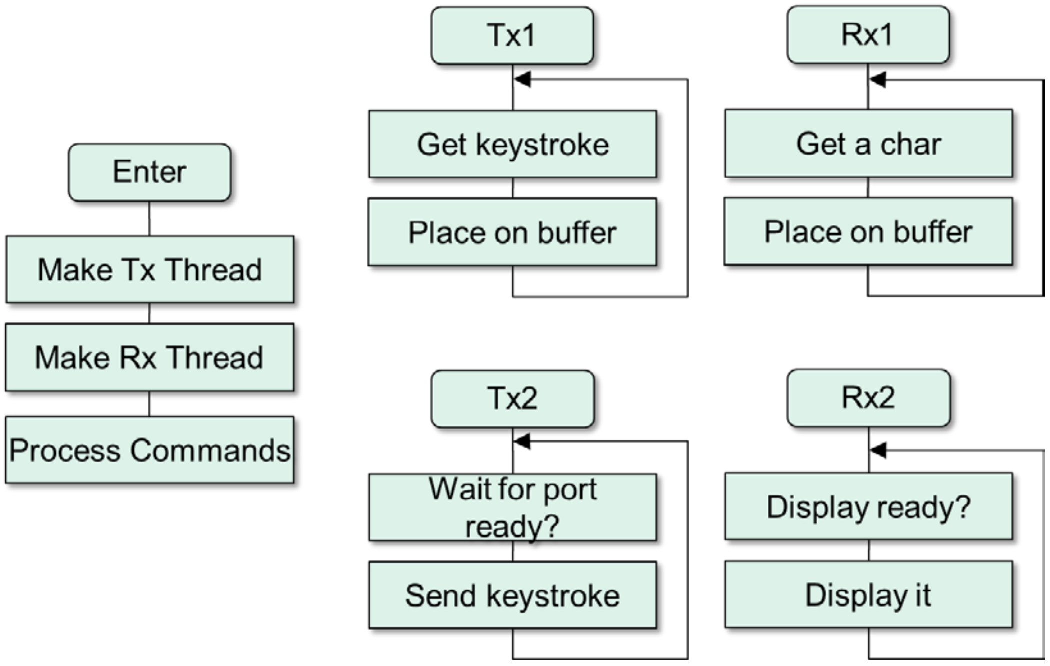


Figure 5. A terminal emulator with double-buffering.

Thread Priorities

OS/2 is a priority-driven multithreaded system. Each system tick, it runs a scheduler which decides which thread to run next, and a dispatcher which actually puts that thread on the CPU. The scheduler maintains two lists of threads. The first list is of threads which are blocked - that is, they cannot run because they are waiting for some event (a semaphore to post, a sector to arrive off disk, etc.). This is the majority of threads. The other list is of threads which are able to run - that is, if they were placed on the processor right now, they could execute without waiting for some event.

The list of ready threads is sort into priority order. When the system timeslices, the scheduler checks for events which might cause threads to unblock, and if so moves the unblocked threads into the ready list at the correct position in the list. Once that has been done, it takes the highest-priority thread off the top of the list and dispatches it.

Notice that if one particular thread is the highest priority and is always able to run, then the system runs it - again and again. It doesn't matter if it's at priority 27 and there's a thread at priority 26 just below it; it gets all the CPU time it needs until it blocks, terminates or drops its priority. However, if there are two threads at priority 27, then the system will round-robin between them.

Priority Classes

There are four priority classes. Each class has 32 levels, ranging from zero to 31.

The top priority class is Time Critical Class. Threads in time-critical class are used to respond in urgent situations. You might think that if you use a time-critical class thread in your application that it will run to the exclusion of everything else in the system, but of course, time-critical class threads spend most of their time blocked and very rarely, if ever, run. However, if a time-critical class thread is say, blocked waiting on a semaphore, you can pretty well guarantee that when that semaphore posts, the thread will run on the very next timeslice, as it will be the highest priority in the system.

Please don't create a time-critical class thread with priority 31 - nothing else could ever get above it, which could cause problems for other programmers. Similarly, priority zero should be avoided.

The next class down is Fixed-High Priority Class, also known as Server Class. This is meant to be used by application code which should run at a higher priority than regular user applications. For example, consider a bank branch. Each teller has a machine running various retail banking applications, but in addition, the end machine is doing double duty: it is running both teller applications and also an SNA gateway, with the other teller machines linked to it via token ring and it in turn providing an SDLC line to a mainframe.

Now, in this scenario, you don't want the teller at the end machine to cause everyone else to wait for mainframe access while he quotes the repayments on a mortgage. So, the SNA gateway would be written to use Fixed-High Priority Class for much of its functionality.

Then come Regular Class threads. This class is where most application threads reside. However, Regular Class is different in one important respect: dynamic priority variation.

The OS/2 'design religion' (as it was termed at Microsoft) strongly differentiates OS/2 from earlier minicomputer operating systems (read UNIX, VMS, etc). Those OS's were designed for a multiuser environment with the goals of maximising throughput and fairness. OS/2, by contrast, is a single-user system, and so the accent is on maximising utility to the user and responsiveness. As a consequence, OS/2 gives a priority boost to the threads of the foreground process, so that if it has ready threads, they will run. After all, the foreground process is the one the user is interacting with and he presumably does not care if something in the background takes a relative priority hit.

As a consequence, OS/2 does not divide CPU time fairly amongst processes or threads. But it does provide responsiveness similar to the DOS environment. On occasions when I have been able to test applications which have the same code base on both OS/2 and DOS (Lotus 1-2-3

3.0 is a good example) the OS/2 version has provided responsiveness similar to the DOS version, almost regardless of what is running in the background.

The scheduler will also give a priority boost to threads that are doing a lot of device I/O but are not using much CPU time to do it (e.g. the spooler). Boosting these threads will hammer a peripheral device that much harder but will have negligible impact on other applications.

But there are pitfalls to this approach. For example, what if the user hits the recalc key with a large spreadsheet application in the foreground (a friend of mine tells tales of spreadsheets which take 48 hours to recalculate! These are goal-seeking spreadshets which attempt to minimise tax on large capital projects). Because the spreadsheet is the foreground application, it gets a priority boost. Because recalculation involves no I/O (paging excepted) the recalc thread will never block and will always be ready to run. As a result, whenever the scheduler timeslices, it will be the highest priority and will always run, to the exclusion of all other regular class threads!

In order to get round this problem, the system uses the `CONFIG.SYS` parameter `MAXWAIT`. With `MAXWAIT` set to the default of 3, a thread which is undergoing starvation (i.e. not able to run because another thread is getting a priority boost) will itself receive a boost after three seconds, will briefly run and then be dropped back to its previous priority. So background application threads will at least crawl towards completion. Perhaps more importantly, this mechanism will break any deadlocks which might arise between threads in a foreground and background application (though such a deadlock is difficult to conceive, stranger things have happened).

An alternative mechanism, for certain situations, would be to disable the dynamic priority variation, with the `CONFIG.SYS` line `PRIORITY=ABSOLUTE`. This would only be used in dedicated systems, however.

Finally, there is Idle-Time Class. Idle-time class threads are used for low-priority background tasks. For example, way back in the early eighties, I used to run a program to generate Mandelbrot sets, which, without the aid of a maths coprocessor, required five days to run. Being unwilling to dedicate a machine to this program, I ran it under Concurrent CP/M, at a low priority. All the real work on the machine ran at a higher priority and always got the processor as required, but when the system was idling (probably 95% of the time during office hours and 100% overnight) the Mandelbrot program ran.

More serious applications for idle-time threads include running background diagnostics, for example memory testing. OS/2 is being used on larger and larger machines, for mission-critical applications. For example, the machine I am typing this on has 64 MBytes of RAM and also runs DB2/2 maintaining customer accounts details - a crash on this or a similar server can be very expensive.

One project I've been meaning to write since OS/2 1.0 appeared (I already had it under MP/M and Concurrent CP/M) is a background memory tester. This would grab a 64 KByte segment of memory (using the `PhysToVirt` device driver helper function) and test it thoroughly - walking bit tests, address line inversion, checkerboard tests and so on. If the segment passes the tests, it is released and the program marches on; but if it fails, the memory is locked down so that it cannot be used by another process, a log entry on disk is generated and a popup window warns the user to call a technician. So many good ideas, and so little time ! - but with the advent of Pentium system boards with no parity checking on the SIMMs, maybe I ought to bump the priority on this little utility.

Have you ever wondered how the Pulse utility works? There is no widely-documented API for extracting CPU time versus real time as on UNIX (although they do exist and are used by the IBM SPM/2 utility), so the information must be deduced. Utilities like pulse.exe work by having two threads running. The first runs at a low priority in idle-time class, incrementing a counter. That's all it does, just loops around, incrementing a counter. The main thread, which is regular class, hangs off a timer, and once a second or so it samples the counter and subtracts the previous value.

Now, if the system is heavily loaded (100% CPU utilisation) the idle-time class thread will not have run, and so the difference between samples will be zero. So the main thread draws a graph at 100%. If the system has nothing else running, there will be a huge difference between samples - the graph shows 0%. And if the system is somewhere in between, the idle-time class thread will have run some of the time, and the graph will be somewhere in between also.

Actually, it's a little more complex, with the counter thread being run at a high priority briefly on startup to establish just how fast the counter will go with no higher-priority threads preempting it. But you get the idea.

Creating Threads

Threads are created using one of several API's. Both OS/2 1.x and 2.x provide the `DosCreateThread()` API's, although they differ considerably in parameters and functionality. In addition, all OS/2 compilers (should?) provide the _beginthread() library function.

DosCreateThread()

The OS/2 1.x version of this API looks like this:

```
BYTE abStack[4096];
TID tidThread;
VOID main() {
     DosCreateThread(ThreadFunc, &tidThread, abStack +
sizeof(abStack));
     .
     .
}
VOID FAR ThreadFunc(VOID)
{
     VioWrtTTY(“Message from new thread\n\n”, 25, 0);
}
```

There are several important things to notice about this API. The first parameter is the thread function (actually, the address of the thread function), while the second is the address of a variable which will receive the thread ID. Finally, we pass in the initial value of the thread's stack pointer. Notice that since stacks grow downwards in Intel processors, the stack pointer is initialized to point to the top end of the stack (which was earlier defined as BYTE abStack[4096]). The stack must be explicitly allocated in our code somewhere; the system does not take care of it automatically.

More importantly, now look at the thread function itself. First, its return type is VOID. Why? Because it never returns, of course - it's a thread, not a function, and the caller does not wait for a returned value! Secondly, it's FAR. Why? Surely it's compiled in the same segment as the main function? Yes, but it's not called from the main function. Instead, it's called from the operating system, so it must be defined as FAR. Finally, notice that we cannot pass any parameters. At least not officially, but there is a little trick - you could define it as taking some parameter (a pointer is safest) and then stick the parameter at the top end of the stack and initialise the stack pointer as abStack + sizeof(abStack) - sizeof(passed type). Tricky, but it can be done.

OS/2 2.x and Warp make life a lot easier. Here's the newer version of the `DosCreateThread()` API:

```
TID tidThread;
struct _threadarg{ . . .} threadarg;
ULONG ThreadFlags;
VOID main() {
         .
     DosCreateThread(&tidThread, ThreadFunc, &threadarg,
ulThreadFlags, STACKSIZE);
         .
}
VOID ThreadFunc(VOID *)
{
     WinSetWindowText(hwnd,"Message from new thread");
}
```

A number of improvements have been made. First, note the reordering of the parameters: the address of the thread ID is passed in first, then the thread function. There's a new parameter allowing a pointer to a user-defined structure to be passed in, thereby allowing parameters to be passed to the thread function. ulThreadFlags really only has two bits of interest: the least significant bit, when set, starts the thread in a suspended state, while its neighbour, if reset, makes the system perform committment of the thread's stack (if set, the stack is precommitted). Finally, we can pass in a stack size, and the system will automatically manage the thread stack for us.

The thread function itself is still `VOID`, since it doesn't return, but it's no longer `FAR`, and we can pass it a parameter, defined as `VOID *`.

This version of DosCreateThread() is much easier to use and more versatile.

_beginthread()

Then there's the `_beginthread()` library function. It basically looks like this:

```
_beginthread(ThreadFunc, pStack, usStackSize, pParms);
```

which at first glance is somewhere between the OS/2 1.x and 2.x versions of `DosCreateThread()`. It allows passing in a `NULL` pointer to a stack, so that the system will automatically allocate and free the stack, and allows passing of a parameter structure.

`_beginthread()` is supported in OS/2 1.x, by Microsoft C and other compilers, and is popular in OS/2 1.x code because of its additional functionality over that provided by `DosCreateThread()`. However, it is frequently used in OS/2 2.x and Warp code, too. Why does the compiler library provide similar functionality to the OS API?

I think I'm right in saying that all compilers for OS/2 provide two sets of runtime libraries. One set is for use with single-threaded programs while the other set is for multithreaded. You see, multithreaded systems require that functions be reentrant, that is, capable of being entered by two threads simultaneously with no ill effects. Take a look at this short snippet:

```
1: int badfunc(int a, int b)
2: {
3:     static fubar;
4:
5:     fubar = 2 * a;
6:     return fubar / b;
7: }
```

OK, it's a stupid function, but I'm trying to make a point here, right? Suppose that thread 1 calls this function, passing in a = 5 and b = 7. Line 5 sets fubar to 10, but just then the system timeslices, and another thread runs. Now, thread 2 calls fubar, with a = 45 and b = 3. It sets fubar to 90 - but notice that because this fubar is static, it's the same fubar as thread 1 was using, and it loses the previous value. Thread 2 now calculates a return value of 30 (correct) and carries on its merry way. Some time later, the system timeslices again and resumes thread 1, which returns a value of fubar=90 / 7 or 12 (wrong! should be fubar = 10 / 7 or 1).

The introduction of statics is a sure-fire way to make a function non-reentrant. If only automatic variables are used, then all values are on the stack, and each thread has its own stack, so no problem, right?

Well, only one problem - performance. Sometimes using statics can simplify and speed up code. So, for performance reasons, the compiler vendors do one library, in which some functions are non-reentrant and therefore only safe in single-threaded processes. But they also do a second library using reentrant code so that the functions are thread-safe, albeit with some performance degradation. The functions affected are usually the more complex ones like `printf()` and `scanf()` and the various file-handling functions. Often, functions like `strcpy()` and `strcat()` are so simple that they are written in assembler using purely registers and the stack and so are reentrant anyway. Consequently, if you know what you are doing, with the aid of your compiler run-time library documentation, you can sometimes use the single-threaded library functions in a multi-threaded program.

But you won't be able to use `_beginthread()` in that case, because it is only found in the multi-threaded libraries. There's another gotcha. The runtime library has to perform error-handling on a per-thread basis, and this requires certain variables and system structures to be initialised on a per-thread basis. This is done by `_beginthread()`, and obviously `DosCreateThread()` cannot do it, since the system has no idea which C runtime library your program was compiled with.

In general, use `_beginthread()` when your program is going to be performing I/O using the C library functions. If your application is written to PM, and your file-handling is done through the `DosRead()`/`DosWrite()` API's, then you are safe to use `DosCreateThread()`, since the operating system is obviously thread-safe.

Priority Variation

Threads are normally created with an initial priority of 15 in the regular class. From there, they can be varied by using the `DosSetPriority()` API:

```
#define INCL_DOSPROCESS
#include <os2.h>

ULONG     scope;    /*  The extent of the priority change. */
ULONG     ulClass;  /*  Priority class of a process. */
LONG      delta;    /*  Change to apply to the current base priority
level of the process. */
ULONG     PorTid;   /*  A process identifier (scope == PRTYS_PROCESS
or PRTYS_PROCESSTREE) or a thread identifier (scope ==
PRTYS_THREAD).*/
APIRET    ulrc;     /*  Return Code. */

ulrc = DosSetPriority(scope, ulClass, delta,
          PorTid);
```

Notice the scope parameter, which allows the programmer to change the priority of a single thread in the current process, all threads in the current process, or all threads in the children of the current process. You cannot change the priority of a single thread in a child process - the programmer of that program set the relative priorities of his threads to work correctly and you should not be changing them. The priority class is a symbolic constant (`PRTYC_NOCHANGE` (0), `PRTYC_IDLETIME` (1), `PRTYC_REGULAR` (2), `PRTYC_TIMECRITICAL` (3) or `PRTYC_FOREGROUNDSERVER` (4)). Notice also the delta - a signed `LONG` value. Thread priority is not set absolutely, but as a delta relative to the current value.

Resuming / Suspending Threads

Thread Termination

Why Threads Are Important

Presentation Manager is a message-passing system. There are multiple queues, but their operation is synchronous - the system will not read the next message until a winproc has returned from processing the previous one. If that winproc does not return, other windows will not get any messages.

This is what happens under Windows, and causes display of the hourglass mouse pointer. However, under OS/2 programmers are advised that if processing a message will take more than 1/10th second, they should do the processing in a second thread, and let the main thread return from the winproc so that more messages can be dealt with.

Actually the famous one-tenth second rule was relaxed in the early days, as a 20 MHz 386 required more than one-tenth of a second to do anything. However, it was reinstituted a few years later, and at the ColoradOS/2 Conference in 1993 most developers agreed it should be amended to be one twentieth of a second to reflect more modern expectations of performance and responsiveness.

Note also that even on modestly powerful machines (486/33) even _beginthread() takes more than a twentieth of a second!

To see the difference compare the Windows and OS/2 versions of an application which has been correctly written to use multiple threads under OS/2 (DeScribe or PageMaker, for example). The Windows version will often display the infamous hourglass pointer, and at that point, other windows become unresponsive. The OS/2 version will do this less often, and even when the hourglass (SYSPTR_WAIT) is being displayed, other windows will remain responsive and other applications can be used.

Thread Types

Finally, some notes on thread types in the Presentation Manager environment. Conceptually, there are two types of threads: message queue threads and non-message queue threads.

Message queue threads are any threads which create a message queue. A message queue is necessary if a thread is going to create a window. However, once a thread has a message queue it is bound by the one-tenth second rule.

Non-message queue threads do not create a message queue. However, they now cannot create windows (since they have no queue to read messages for the window). But such a thread can paint in a window, using `WinBeginPaint()`, `WinDrawText()` and various Gpi calls (note, however, that the thread must call `WinInitialize()` before doing so). A non-message queue thread also cannot call `WinSendMsg()`, but it can call `WinPostMsg()`, for example to signal completion of a task (like painting a window). See the section of these notes on the Model-View-Controller paradigm for more information on using a secondary thread to paint a window.

Non-message queue threads are free from the one-tenth second rule, since they do not have a message queue.

There is one other exception to the one-tenth second rule, in the form of object windows. A thread which creates and runs an object window is exempt from the rule, since the object window plays no part in the PM user interface - the only system-generated messages it will receive are `WM_CREATE`, `WM_DESTROY` and `WM_QUIT`. All other messages are user-defined.

Interprocess & Interthread Communication

The OS/2 control program goes to great lengths to isolate processes and provide inter-process protection. However, much of the value of a multitasking system comes through the ability to assemble program systems, in which multiple programs cooperate and the whole is greater than the sum of the parts. This will require formal mechanisms for communicating over the inter-process firewalls while still providing protection.

OS/2 therefore provides a rich array of inter-process communications mechanisms. It is often said that the problem for an OS/2 programmer is not finding a way to do something, it is selecting one of the three alternative ways the system provides, and nowhere is this more true than in IPC.

Pipes

Anonymous pipes are typically used from the command line. For example, the command

```
DIR | SORT | MORE
```

connects the output (stdout) of the `DIR` command to the input (stdin) of the `SORT` command, and `SORT`'s stdout to the stdin of `MORE`. DOS fakes this through temporary files, but OS/2 uses genuine in-memory buffers to link the processes together.

The natural record to pass through an anonymous pipe is a line of text, but binary transfers are possible (although unusual with filter programs).

Although `command line redirection` is the most common technique, it is also possible to use anonymous pipes programmatically. Remember that a child process inherits its parent's environment, including file handles, so that it is possible for a parent process to redirect stdin and stdout, use `DosExecPgm()` to spawn a child and then redirect stdin and stdout again so as to feed data to the child and get it back again.

The relevant API's to investigate are `DosCreatePipe()`, `DosOpen()` (check out the `OPEN_FLAGS_NOINHERIT` flag, which stops a child from inheriting a file handle), `DosDupHandle()`, `DosSetFHState()`, and `DosQueryFHState()`.

Named Pipes

Named pipes were originally developed by Microsoft as a client-server communications protocol in MS LAN Manager 1.0. IBM wanted nothing to do with named pipes, preferring to promote SNA LU6.2 APPC (Advanced Program - Program Communication) as a platform-independent client-server API. The problem was, the Communications Manager required for APPC provided no file or print sharing services for PC workgroups and was horrendously slow as well, so that IBM was forced to licence MS LAN Manager and remarket it, with additions, as IBM LAN Server. However, they still refused to document the named pipes API, although they did not remove it from the code.

In order to do an end-run around IBM, Microsoft moved the named pipe code out of LAN Manager and into the OS/2 kernel with OS/2 1.1, so that IBM was forced to support the API. (The plot thickens: Microsoft subsequently dumped named pipes in favour of sockets and RPC, but poor old IBM is still supporting a protocol that was forced on them rather than inconvenience their customers. That's life, as they say.)

Named pipes actually provide quite a neat and useful protocol, which is easy to write to and gives good performance. Essentially, named pipes allow pipes to be named and extended across the network, although they do work standalone in a single machine. Network operation requires a LAN Server or similar server, however. Whether a pipe is local or remote is pretty much transparent as far as the coding is concerned; remote pipes will be accessed using a UNC-name which as the server name as the first part, whereas local pipes just use the pipe name.

Named pipes can be inbound, outbound or full duplex, in which case they provide a virtual circuit abstraction. A single named pipe can be used and reused by multiple clients, and multiple pipes can be pooled under a single name. On a network, access to a server pipe is subject to user logon permissions, so that security is implemented.

For the most part, pipes are dealt with as though they are some kind of interactive file - they use the same handles (`DosQueryFHState()` can distinguish whether a handle is for a pipe, a file or a device) and the same open, read, write and close calls.

Here is the skeleton of a named pipe server, to be run at a server called \\NETPC:

```
DosCreateNPipe("\\PIPE\\DBMS", &hpipe);
while(more) {
     DosConnectNPipe(hpipe);      /* await client*/
     DosRead(hpipe, &req);        /* read request */
          .                  /* process request */
     DosWrite(hpipe, &resp);      /* send response */
     DosDisconnectNPipe(hpipe);   /* close client */
}
DosClose(hpipe);
```

Notice that in this case, the server disconnects from the pipe after each transaction so that a difference client can attach to it and reuse it. Not all parameters have been shown here - I'm concentrating on the overall logic.

At the client, the code will look something like this:

```
DosOpen(“\\\\NETPC\\PIPE\\DBMS”, &hpipe, . .);
DosWrite(hpipe, &req);
DosRead(hpipe, &resp);
DosClose();
```

or like this:

```
DosOpen(“\\\\NETPC\\PIPE\\DBMS”, &hpipe, . .);
DosTransactNPipe(hpipe, &resp,. ., &req, . . .);
DosClose(hpipe);
```

or even like this:

```
DosCallNPipe(“\\\\NETPC\\PIPE\\DBMS”,&resp, . ., &req, . .);
```

Now do you see what I mean about OS/2 RPC being an embarrassment of riches?

The first technique allows an unbalanced number of requests and responses. For example, an SQL query could be sent to a server, followed by a number of reads to get back table rows until a ‘no more data’ response is encountered. The other techniques require a matched pair of request/response. The second technique is probably more efficient than the third in the case where a client is going to attach to a server and perform a number of transactions, at least in terms of CPU cycles. However, it may not be as efficient in terms of network packets, another important consideration. On a NETBIOS LAN, the first client will require twelve packets across the LAN, the second will require four and the third technique will require two packets only.

There's another important distinguishing factor in the first approach. Can you see what it is?

The answer is that it uses no OS/2 or named-pipe-specific API’s. In other words, it can be used with any programming language or application that can open a file, read and write it, and close it. This includes DOS applications - the DOS LAN requester in MS LAN Manager and IBM LAN Server is able to deal with named pipes. The second and third techniques use named pipe calls, which are supported by the appropriate network developers’ kits, but of course are linkable from C code or languages which can support the C calling convention only, which rules out interpreted BASIC programs, Lotus 1-2-3 and Excel macros, and so on. The first technique, however, will *always* work.

Now, this is really useful when you are called upon to perform some systems integration, linking DOS and Windows programs together. With a little ingenuity you can construct an OS/2 named pipe server in C code, and then write application code to link together multiple DOS and Windows applications on a single OS/2 machine (or even a network). This can even be done using REXX code, if you want something quick and a little dirty.

The syntax for DosCreateNPipe is:

```
 #define INCL_DOSNMPIPES
 #include <os2.h>

 PSZ       pszName;   /*  The ASCIIZ name of the pipe to be opened.
*/
 PHPIPE    pHpipe;    /*  A pointer to the variable in which the
system returns the handle of the pipe that is created. */
 ULONG     openmode;  /*  A set of flags defining the mode in which
to open the pipe. */
 ULONG     pipemode;  /*  A set of flags defining the mode of the
pipe. */
 ULONG     cbOutbuf;  /*  The number of bytes to allocate for the
outbound (server to client) buffer. */
 ULONG     cbInbuf;   /*  The number of bytes to allocate for the
inbound (client to server) buffer. */
 ULONG     msec;      /*  The maximum time, in milliseconds, to wait
for a named-pipe instance to become available. */
 APIRET    ulrc;      /*  Return Code. */

 ulrc = DosCreateNPipe(pszName, pHpipe, openmode,
          pipemode, cbOutbuf, cbInbuf, msec);
```

The `openmode` parameter allows the programmer to specify inbound/outbound/full duplex, will the pipe be inherited by children, and enable write-through cacheing. The `pipemode` parameter controls blocking (does e.g. `DosRead()` block if no data is available or return immediately)and sets the pipe mode to be either byte mode (read or writes a stream of bytes) or message mode (read or write a message, with the message length in the first two bytes of the message). The low-order eight bits of this parameter is used to specify an instance count for a pipe pool.

Other pipe-related functions include:

- `DosCreateNPipe()`
- `DosConnectNPipe()`
- `DosDisConnectNPipe()`
- `DosTransactNPipe()`
- `DosCallNPipe()`
- `DosPeekNPipe()`
- `DosQueryNPHState()`
- `DosQueryNPipeInfo()`
- `DosQueryNPipeSemState()`
- `DosSetNPHState()`
- `DosSetNPipeSem()`
- `DosWaitNPipe()`

Shared Memory

OS/2 supports two primary forms of shared memory: 'giveaway' shared memory and named shared memory. The two are distinguished by the referent passed between processes in order to gain access to the shared memory. In the case of giveaway shared memory this is a pointer (in 1.x, a selector) - but this causes a chicken-and-egg situation to arise. Pointers and selectors are runtime entities and their values cannot be known at compile time, so that they must be passed between the processes at runtime using some other form of interprocess communications.

An obvious candidate for this is Presentation Manager user-defined messages - just define a message and pass the pointer or selector in one of the mparams. Of course, this requires that the target window handle must be known in advance, possibly by identifying the target process from its entry in the switch list, and `...`

Named shared memory, by contrast, is referred to by name, and the name can be agreed on at compile time, possibly defined as a symbolic constant in a shared header file.

Remember, what two consenting adult processes do in shared memory is entirely their business! You may need to protect data structures in shared memory with a semaphore to ensure correct updating, and rigidly enforce protocols as to who allocates and who frees memory and data structures such as linked lists.

Giveaway Shared Memory

Giveaway shared memory can further be subdivided into two types: giveable and gettable. The two differ only in the sequence of API's used. Both are created using `DosAllocSharedMem()` with either the `OBJ_GIVEABLE` or `OBJ_GETTABLE` flags.

To allocate and use gettable memory:

```
DosAllocSharedMem(&pv, NULL, size, OBJ_GETTABLE);
```

Pass pointer to other process

Other process calls `DosGetSharedMem()` to validate memory object address

To use giveable shared memory, here's the sequence:

```
DosAllocSharedMem(&pv, NULL, size, OBJ_GIVEABLE);
DosGiveSharedMem(pv, pidTarget, flags);
```

then pass `pv` to Target process, which need not call `DosGetSharedMem()` to validate before using

Named Shared Memory

Named shared memory is also allocated using `DosAllocSharedMem()`, except that now a pointer to a name is specified as the second parameter. The second process gets access to the memory using `DosGetNamedSharedMem()`.

Queues

Queues can be thought of as 'structured pipes'. Queues exist in the filesystem namespace as '`\QUEUES\NAME`'. Any process can write to a queue, but the only process that can read a queue is the process that created it. The reader can read the queue in FIFO or LIFO sequence (queue or dequeue / stack), or in priority sequence. It can also peek in the queue, and perform a non-destructive read of any element.

Queues are high performance: the reason is that large blocks of data are not really passed through the queue. Instead, the data to be passed is actually established in shared memory, and then a pointer to the data is passed, as part of a small structure, through the queue. This avoids the overhead of copying large data structures onto and off the queue, thus improving performance. In essence, therefore, the queue is actually a linked list of up to 3192 items.

Beware: OS/2 1.x and 2.x/Warp support almost identical queue API's. One might be tempted to port a pair of 1.x applications which use a queue to communicate, by porting first one of the pair, and using the 1.x server to test the prototype 2.x client. This will not work, as the 16-bit queues and 32-bit queues are not compatible and in fact occupy different namespaces.

The queue API's of interest are:

- `DosCreateQueue()`
- `DosOpenQueue()`
- `DosReadQueue()`
- `DosWriteQueue()`
- `DosPurgeQueue()`
- `DosCloseQueue()`
- `DosQueryQueue()`
- `DosPeekQueue()`

Semaphores

Semaphores have been implemented with two quite different architectures in OS/2 1.x and 2.x/Warp. We shall deal first of all with the OS/2 1.x semaphore mechanism, as a means of intriducing some of the concepts and also to assist 1.x developers and those who will have to port 1.x code, and then progress to 2.x semaphores.

OS/2 1.0 had two types of semaphores: RAM sempahores and system semaphores. The RAM sempahores exist in memory inside a process, and are therefore only of use for intra- (not inter-)process communications. Why might a process want to communicate with itself? Answer: threads.

OK, if two threads need to communicate, why not just define a shared variable and let them use it to signal what they are up to? It won't work. Let's see why.

Take a hypothetical application which has several threads, each of which might send something to the printer. But the printer is a serially reusable resource - only one thread should be able to print at a time, otherwise the output will be jumbled (of course, at the application level the spooler subsystem solves the problem).

OK, so we define a shared variable, `ULONG ulPrinterInUse = 0x00000000`; if a thread is printing it must set the variable to `0xffffffff` and then reset it to `0x00000000` afterwards. Of course, before starting to print, the thread must test the value of the ulPrinterInUse variable. So, here's what can happen in the compiled machine code:

Thread x needs to print something. It moves the value of `ulPrinterInUse` into a processor register and then executes a `JZ` (jump if zero instruction). The printer is not currently in use, so it jumps to a subroutine which will `DEC`rement the value in the register (to `0xFFFFFFFF`) and then write this new value back - but before it can do so, the system timeslices. *Remember, OS/2 is a preemptive multitasking system!*

Now, thread y comes along and does the same thing: it tests the value of `ulPrinterInUse` by means of a `JZ` instruction, finds it is zero and branches accordingly, then decrements the value, writes it back and starts printing. A little later, thread y is preempted, the system timeslices and thread x is back on the machine. It resumes where it left off, decrementing the register and writing back `0xFFFFFFFF`, then starting to print. Ouch!

The use of a shared variable has not protected us against simultaneous use of a resource, because the act of testing, and if clear setting, a flag variable must be atomic - that is, indivisible or uninterruptible. Conventional code is not.

We could deal with this problem by making these few lines of code into a critical section; we only need to add `DosEnterCritSec()` and `DosExitCritSec()` API calls around the code. When a thread enters a critical section, all other threads in the same process are suspended until it exits again. In other words, a critical section suspends multithreading within a process. Warning! Don't crash inside a critical section - it gets really ugly, really quickly! Critical sections should only be used after extensive testing - if there's an easier alternative, use it.

Another problem is, what will you do if your thread discovers a resource is not available? Loop around, polling the value of the shared variable, until it does become available? I don't think so. We're already agreed that polling is A Very Bad Thing in a multitasking environment; what we really need is a way for the system to suspend the thread until such time as the resource becomes available and the 'shared variable' is reset.

And so, RAM semaphores were born. The provision of RAM semaphores in the system eliminates the need for messy and potentially fatal critical sections, and since the system knows about them, looping and polling in application threads is also not necessary. The system can block a thread until such time as a resource becomes available. However, you should note that RAM semaphores remain words of memory in an application's address space, and are therefore subject to the usual concerns of being overwritten by rogue pointers or other bad programming practices.

System semaphores, by contrast, exist outside any application's address space. They are therefore safer, and are suitable for inter-process communications. They are also named: remember, resources that are referred to by a handle require some other form of IPC to pass the handle around in a kind of chicken-and-egg puzzle, while names are known at compile-time, making coding a lot easier.

However, system semaphores are inevitably slower than RAM sempahores.

So, in OS/2 1.2, so-called fast-safe semaphores were introduced. These combine the speed of RAM sempahores with the safety of system semaphores.

In OS/2 2.x and Warp, the semaphore architecture was changed substantially, and looks more like the macros used for the purpose in the VM and MVS operating systems. There are three types of semaphores, basically categorised according to their purpose, and each of these can be private or shared (rather like RAM or System) and can be unnamed or named.

Event semaphores are used to signal the occurrence or completion of some event. Typically, one thread creates a semaphore and resets it. A second semaphore then opens the semaphore and waits on it. The first thread posts the semaphore and the second thread then unblocks and resumes execution.

Mutex semaphores are used for mutual exclusion - that is, protecting some resource against simultaneous access by two or more threads (or processes). Typically, thread one creates the mutex semaphore to protect some resource. Other threads then request the mutex; the first one to do so will own the resource and return from the request, while others will block. Once the first thread releases the mutex, another will unblock and can use the resource, and so on.

Finally, muxwait semaphores are compound semaphores which consist of up to 64 event or mutex sempahores (the types cannot be mixed). A muxwait can be set up to resume a thread when any one of n resources becomes available, or when all n resources are available, or when events occur, and so on.

Let's look at that again, this time in slow motion. See Figure 6. In this example, the first thread, on the left, creates an event semaphore (the process that creates an event semaphore does not need to open it - it already has its handle). It then resets the semaphore; a reset semaphore is like a red traffic light - threads which encounter it will stop and wait, as can be seen in the three threads at right. Each of them opens the semaphore (not necessary if they are part of the same process) and then waits on the event semaphore. They all block until the main thread calls `DosPostEventSem()`, at which point they are all released and resume execution.

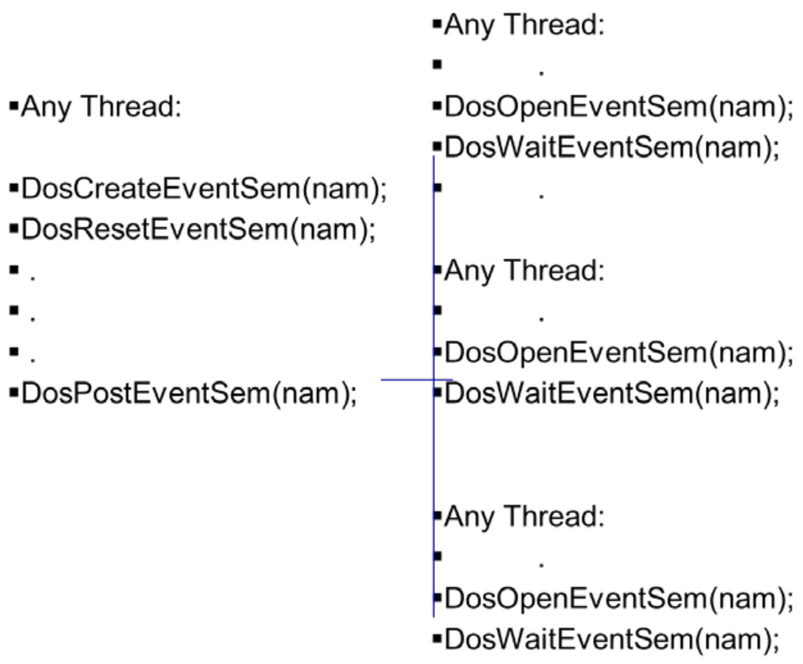
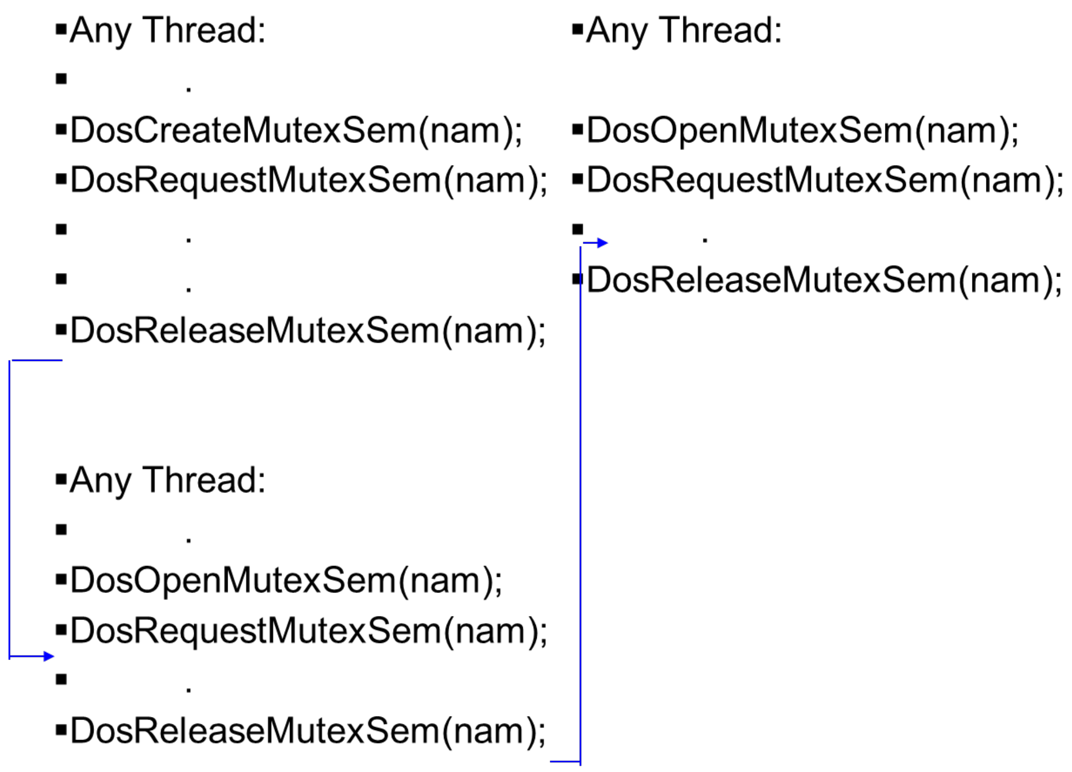


Figure 6. Operation of an event sempahore.

Now, let's look at a mutex semaphore. See Figure 7. Here, thread one creates a mutex semaphore and immediately requests it. Of course, the request succeeds, and it now owns the associated resources, which it proceeds to manipulate. Meanwhile, thread 2 opens the mutex and then requests the mutex (or more properly, the resource the mutex protects). Of course, it blocks in the request call and so the thread blocks until such time as thread one calls `DosReleaseMutexSem()`, at which point thread 2 resumes and can now manipulate the resource. Meanwhile, back at the farm, thread three has also opened the mutex and called `DosRequestMutexSem()`, so it blocks until such time as thread two is finished with the resource and calls `DosReleaseMutexSem()`.



```
▪Any Thread:
▪        .
▪DosCreateMutexSem(nam);
▪DosRequestMutexSem(nam);
▪        .
▪        .
▪DosReleaseMutexSem(nam);

▪Any Thread:
▪        .
▪DosOpenMutexSem(nam);
▪DosRequestMutexSem(nam);
▪        .
▪DosReleaseMutexSem(nam);

▪Any Thread:

▪DosOpenMutexSem(nam);
▪DosRequestMutexSem(nam);
▪        .
▪DosReleaseMutexSem(nam);
```

Figure 7. Operation of a mutex semaphore.

It's quite straightforward really.

OS/2 File Handling

OS/2, like UNIX, has no knowledge of the internal structure of files, nor does it care particularly about file types, with a couple of exceptions: command (program) files must have a filetype of .COM, .EXE or .CMD. The latter type is a batch file or REXX procedure.

The operating system regards a file as simply a stream of bytes, located on a serial or random access device. The file can be opened read and written, and any number of bytes may be read or written, up to the limits of the medium or the ultimate limits of the operating system (e.g. 2 GB max file size). OS/2 does not understand the difference between, say, text files and binary database files or spreadsheet files; it has no concept of records and fields or rows and columns. These are the domain of the application software.

Files are distinguished by their names, which must be unique within a directory (that is to say, it is possible to have identically-named files in two different subdirectories or on two drives). Different file systems (FAT, HPFS, CDFS) have different file-naming conventions, and it is important that programs should be written to be as portable as possible. For this reason, OS/2 programs which ask the user for filenames should allocate or otherwise reserve a buffer of at least 255 bytes in size for the filename, rather than the 64 bytes used by DOS.

The file name is only used to open the file; from that time on it is referenced by a file handle (a variable which is typedef'ed to be of type HFILE). Ultimately, of course, a file handle must resolved to one of the C scalar types such as int or pointer to structure, but the programmer should consciously avoid knowing what a HFILE really is, in order to avoid making assumptions which lead to non-portable code. For this reason, handles are also referred to at Microsoft as 'magic cookies', since no-one knows what they are, either.

Opening Files

There are two major function calls in OS/2 1.X for opening files.

DosOpen()

```
USHORT DosOpen(pszFileName, phf, pusAction, ulFileSize, usAttribute,
usOpenFlags, usOpenMode, ulReserved)
```

is the basic function call. OS/2 1.2 introduced

DosOpen2()

```
USHORT DosOpen2(pszFileName, phf, pusAction, ulFileSize, usAttribute,
usOpenFlags, ulOpenMode, peaop, ulReserved)
```

which supports the use of extended attributes. In OS/2 2.0 and later, the function is called `DosOpen()`, but it takes the formal parameter list of `DosOpen2()` above.

Let's examine the major parameters.

First, remember that in OS/2, many parameters passed to the function are not just input to the function, but may also be outputs or 'returned' values. Since C passes parameters by value (i.e. it copies the contents of variables onto the stack and then works with these copies of the original data) any changes to parameters in a function do not affect the original variables and therefore values cannot be 'returned'. Consequently, such variables are not passed directly - instead, a pointer to the variable is passed and these are commonly prototyped as pointers to variables. However, although the parameter is prototyped as a pointer, that does not mean that the corresponding variable is a pointer. For example, take the file handle for these function calls. It would actually be declared as a file handle, and then its address taken to create a 'pointer constant', like this:

```
HFILE infile;
USHORT rc;

rc = DosOpen(argv[1], &infile, . . .);
```

Now, back to the parameters:

`pszFileName` is a pointer to the null-terminated string that specifies an existing file or will create a new file. Wildcards are not permitted, obviously.

`phf` is the address of the variable which will receive the file handle 'returned' by the function.

`pusAction` is the address of a variable which will be set to indicate the action the function took. If the function fails, it has no meaning, but if it succeeds, it will be set to one of the following values:

`FILE_CREATED`	File was created.
`FILE_EXISTED`	File already existed.
`FILE_TRUNCATED`	The file existed and was truncated to the specified size.

`ulFileSize` is the new size of the file. This parameter applies only when a file is created or truncated.

`usAttribute` specifies the file attributes. This can be a combination of the following values:

`FILE_NORMAL`	File can be read and written.
`FILE_READ_ONLY`	File can be read but not written.
`FILE_HIDDEN`	File is hidden and does not appear in directory listings.
`FILE_SYSTEM`	File is a system file.
`FILE_ARCHIVED`	File has been archived.

These attributes are changed only if the file is being created.

`usOpenFlags` specifies the actions to take in the case of the file existing or not. It can be one of the following values:

`FILE_CREATE`	Create a new file; fail if it already exists.
`FILE_OPEN`	Open an existing file, fail if it does not exist.
`FILE_OPEN | FILE_CREATE`	Open an existing file or create a new one
`FILE_TRUNCATE`	Open an existing file and change its size.
`FILE_TRUNCATE | FILE_CREATE`	Open an existing file and change its size ot create a new file of that size.

`ulOpenMode` specifies the combination of an access mode and a share mode, plus certain other modes. See below for access and share modes. The other possible values are:

`OPEN_FLAGS_DASD`	Opens a physical drive for direct access. This can be used with the DosDevIOCtl() function to bypass the file system an access sectors directly, e.g. for defragmentation.
`OPEN_FLAGS_FAIL_ON_ERROR *`	Bypasses the system critical-error handler, so that the user never sees error pop-ups when there are file system errors such as bad sectors or drive door open. The application must deal with all errors.
`OPEN_FLAGS_NOINHERIT *`	The file handle is not available to any children. Default behaviour is for children to inherit their parents file handles.
`OPEN_FLAGS_WRITE_THROUGH *`	If this flags is specified, then the system will write data to the device before returning. Without it, data may be cached in the OS buffers and written later.
`OPEN_FLAGS_NO_LOCALITY`	There is no specific information regarding the locality of reference (i.e. the degree of randomness with which the file is accessed).
`OPEN_FLAGS_SEQUENTIAL`	The file is accessed sequentially.
`OPEN_FLAGS_RANDOM`	The file is accessed randomly.
`OPEN_FLAGS_RANDOMSEQUENTIAL`	The file is accessed randomly, but with a degree of sequential access within that, such as reading or writing large blocks of data randomly.
`OPEN_FLAGS_NO_CACHE`	The disk driver should not cache data in I/O operations on this file.

* These flags can also be modified and queried using the `DosSetFHandState`() and `DosQFHandState`() functions.

`peaop` is a pointer to an EAOP structure which defines extended attributes for the file. This parameter only applies to the `DosOpen2()` function call.

`ulReserved` is a reserved value and must be set to zero.

The DosOpen and DosOpen2 functions return zero upon success, or an error code in case of failure.

Creating Files

Files are created by opening them with the `FILE_CREATE` value for the fsOpenFlags parameter.

Reading and Writing

DosRead()

```
USHORT DosRead(hf, pvBuf, cbBuf, pcbBytesRead)
HFILE hf;               /* File to read */
PVOID pvBuf;            /* address of buffer */
USHORT cbBuf;           /* count of bytes in (size of) buffer */
PUSHORT pcbBytesRead;   /* count of bytes actually read */
```

This function reads up to the specified number of bytes into a buffer, and returns zero upon success or an error code.

DosWrite()

```
USHORT DosWrite(hf, pvBuf, cbBuf, pcbBytesWritten)
HFILE hf;                    /* File to write */
PVOID pvBuf;                 /* address of buffer */
USHORT cbBuf;                /* count of bytes in (size of) buffer */
PUSHORT pcbBytesWritten;     /* count of bytes actually written */
```

DosReadAsync()

DosWriteAsync()

These OS/2 1.X functions perform reading or writing in a secondary thread in the kernel, allowing your program to continue performing other processing. DO NOT USE THEM! They are non-portable to OS/2 2.X, their operation is quite specific and limited in value. The same effect, in a more general and versatile form, can be achieved by simply creating a thread which calls DosRead() or DosWrite().

DosChgFilePtr()

This function moves the file's read/write pointer. This points to the next byte to be read or written.

```
USHORT DosChgFilePtr(hf, lDistance, fMethod, pulNewPtr)
HFILE hf;
LONG lDistance;
USHORT fMethod;
PULONG pulNewPtr;
```

The parameter fMethod specifies the location to which lDistance will be relative. It will be one of the following values:

FILE_BEGIN	Start move at the beginning of the file
FILE_CURRENT	Move relative to the current position
FILE_END	Move relative to the end of the file.

The parameter lDistance is signed. Positive values move forward, negative values backward through the file.

DosBufReset()

This function flushes the buffers for the specified file, writes to disk and updates the file's directory entry. The file remains open.

File Handles

DosSetMaxFH()

By default, an OS/2 process can open up to twenty file handles. However, five handles are automatically opened for stdin, stdout, stderr, stdprn and stdaux. In addition, file handles are also used for module (DLL) handles. Many applications will therefore have to exceed the limit of twenty. This is done by using the DosSetMaxFH() function.

```
USHORT DosSetMaxFH(usHandles)
USHORT usHandles;
```

DosSetFHandState()

DosQFHandState()

These functions can be used to set and query a file's inheritance, fail-on-error and write-through flags (see the DosOpen function for details).

DosDupHandle()

Unless the `OPEN_FLAGS_NOINHERIT` flag is turned on, a child process will inherit its parents handles. This is a useful facility, but it does have one drawback: if the child closes the handle as it terminates, it closes it for the parent, too. It is therefore common practice for the parent to create a duplicate handle for a file, then pass the duplicate to the child through (typically) shared memory. The child can manipulate the duplicate handle without affecting the parent.

File Locking and Region Locking

Although OS/2 is often implemented as a single-user standalone system, even a single user may - intentionally or deliberately - invoke an application twice and attempt multiple operations on a single file. When implemented multi-user or on a network workstation, the likelihood of concurrent access to a single file increases further. In order to ensure data integrity, it is important that access to either entire files, or sections of files, is restricted according to some simple rules.

Access Modes

The DosOpen() and DosOpen2() function calls' ulOpenFlags parameter should incorporate one of the following access modes:

`OPEN_ACCESS_READONLY`	Program can only read from file, not write.
`OPEN_ACCESS_READWRITE`	Program can read and write the file.
`OPEN_ACCESS_WRITEONLY`	Program can write to the file, but not read.

Share Modes

The DosOpen() and DosOpen2() function calls' ulOpenFlags parameter should incorporate one of the following share modes:

`OPEN_SHARE_DENYNONE`	Other processes can open the file for any access mode (read-only, write-only or read-write)

`OPEN_SHARE_DENYREAD`	Other processes can open the file for write-only access but they cannot open if for read-only or read-write access.
`OPEN_SHARE_DENYREADWRITE`	The current process has exclusive access to the file. The file cannot be opened by any process.
`OPEN_SHARE_DENYWRITE`	Other processes can open the file for read-only access but they cannot open it for write-only or read-write access.

A process that is writing a log file, for example, might open the log in OPEN_ACCESS_WRITEONLY access mode and OPEN_SHARE_DENYNONE share mode, so that other processes can write to the log file and a monitor process can read it.

DosFileLock()

This function unlocks and locks a region in an open file. It takes, as arguments, a pair of pointers to FILELOCK structures:

```
typedef struct _FILELOCK {
     LONG lOffset;
     LONG lRange;
} FILELOCK;
```

The region is specified as a number of bytes (lRange) starting at position lOffset within the file.

```
USHORT DosFileLocks(hf, pfUnlock, pfLock)
HFILE hf;
PFILELOCK pfUnlock;
PFILELOCK pfLock;
```

If a NULL is passed instead of either of the PFILELOCKs, then the parameter is ignored.

The function will return zero if successful or one of two error codes:
`ERROR_INVALID_HANDLE` or `ERROR_LOCK_VIOLATION.`

Obtaining Information About a File

DosQFileInfo()

DosQFileMode()

DosQAppType()

The Model-View-Controller Approach to PM Programming

A common question that gets asked in OS/2 programming forums and conferences relates to the use of multiple threads to update windows. Programmers who try this often find that they are getting PM resource interlock errors, SYS3175's and other problems.

Another set of problems relate to the timing of when to paint in a PS/window: some programmers will write code like this:

```
case WM_COMMAND:
     switch(SHORT1FROMMP(mp1)) {  // What does the user want done?
          case IDM_UPDATESTUFF:
               // recalculate some stuff
               .
               .
               // Now update the screen
               hps = WinGetPS(hwnd);
               GpiCharString(hps,. . .);
               GpiCharString(hps, . . .);
               // etc
               WinReleasePS(hps);
               return (MRESULT) FALSE;
               break;
          //more of the same follows
     } // end of switch in WM_COMMAND
```

However, as the application becomes more complex, problems start to arise and the code becomes progressively more difficult to enhance or even maintain.

Howver, there is a simple approach to application (and indeed, window) design which will keep you on the right track. This technique is borrowed from object-oriented programming in Smalltalk, which implemented one of the earliest graphical windowing systems on Xerox workstations.

This approach breaks the application into three components, and is called the Model-View-Controller approach.

The Model is the business logic and internal representation of the application. In the case of our simple Lab 2 exercise, it is simply the character array which contains the string to be displayed on the window. In more complex applications, it may be a structure of some complexity, such as a base structure which in turn contains the root of a linked list of other structures (representing lines in a list, spreadsheet cells or graphical objects).

The View is the on-screen representation of the model. In the case of Presentation Manager, it is the on-screen bitmap which occupies the client window area. The way in which we update this is by painting in a presentation space, and the PS therefore forms part of the View, as well.

The Controller is the code which is responsible for updating the View in response to notification of events from the model, and in the context of PM programming, is obviously the Window Procedure - specifically, the code which paints in the presentation space.

How do we apply this to PM program design?

The first thing to note is that presentation spaces are non-reentrant. That is, it is not possible for two threads to paint in or otherwise manipulate a presentation space at the same time. The reason for this is that a presentation space is a static entity. Furthermore, it is modal - at all times, it maintains attributes for the various primitives, such as the current font, current position, current linewidth and so on. It is not possible for two threads to try, for example, to write character strings in two different fonts at the same time.

Consequently, we need to arrange things so that the PS is protected against access by multiple threads simultaneously. The first approach we might try is to use a mutex (mutual exclusion) semaphore to control access to the PS. This will work, and is initially attractive, but again, complexity tends to multiply and a simpler approach is desirable.

The second approach - the MVC approach - is simply to have a single block of code (in more sophisticated designs, a single thread) which access the PS. In other words, only paint the window in one single place. And where should that be? Answer: in the WM_PAINT stub of the window procedure.

The WM_PAINT message is sent to the window whenever any part of the window is invalid, for whatever reason: the window is brought to the foreground, enlarged, restored, maximised, a menu which was obscuring it is dismissed, whatever. Whenever part (or all) of a window was obscured by something else, but is now visible, it will become invalid and will need to be repainted.

As parts of the screen become invalid, the system accumulates them into the 'invalid region' (a region is simply a collection of intersecting rectangles). With the default class style of asynchronous painting, it will wait until the application queue is empty of higher-priority messages (and they almost all are) and then generate a WM_PAINT message.

When, in response to the WM_PAINT message, the application issues a `WinBeginPaint(hwnd, hps, &rectl)` call, PM fills in the supplied `RECTL` structure with the bounding rectangle of the invalid region, and so the application now knows just what area of the window needs to be repainted.

(In the simple applications used here, we simply repaint the entire window, so that the text is remains centered in the window, rather than centered in the invalid rectangle. You might try commenting out the `WinQueryWindowRect(hwnd, &rc)` call from Lab 1 or Lab 2 to see the difference. However, more complex applications can take some time to repaint an entire window, so it is worth while working out exactly what needs to be repainted and just doing that).

The `WinBeginPaint()` function call also causes the system to reset the invalid region to null.

So, it doesn't matter whether the entire window needs to be repainted, perhaps because it was just created, or only a small part of it needs to be updated, perhaps because one corner which was in the background has just come to the foreground. The WM_PAINT logic should always be written to repaint the required area of the window as efficiently as possible.

Given that the WM_PAINT processing can repaint any or all of the window, we now have the bulk of the view logic implemented.

Now, how to deal with changes in the model? In the simple Lab 2 example, the internal representation is a string, which the user can change by selecting a famous (or infamous!) person from the menu, which of course, generates a WM_COMMAND menu.

Now, is there anything to stop the programmer from writing code which immediately paints the window with the new quote? No, nothing, other than the dire warnings issued in this article.

Instead, the smart programmer will think of the MVC approach. Having structured the `WM_PAINT` processing to update the view, the question becomes, how do we get the `WM_PAINT` code to run?

The first impulse of most beginning PM programmers is to post or send themselves a `WM_PAINT` message. However, this will not work - the code does run, but because the system has not added any part of the window to the invalid region, the `WinBeginPaint()` function call returns a null rectangle in the supplied RECTL structure, so nothing gets painted. The trick is to get PM to add the window (or the desired part of it, for efficiency) to the invalid region. A quick look through the Win API's will reveal the `WinInvalidateRect()` function.

To get the entire window to repaint, you can use `WinInvalidateRect(hwnd, NULL, FALSE)` or `WinUpdateWindow(hwnd).` To update just part of the window, first work out the invalid rectangle, then call `WinInvalidateRect(hwnd, &rectl, FALSE).` Another useful function is `WinInvalidateRegion().`

All of these functions add the window, or part of it, to the invalid region. What happens next depends upon the class style or window style of the window. If the window has class style `CS_SYNCPAINT` or window style `WS_SYNPAINT`, then it is a synchronous window, and a `WM_PAINT` message will be generated immediately. The window will be redrawn before the `WinInvalidateRect()` function returns. However, if the window does not have these styles, then it is an asynchronous window. The system waits until the message queue is empty before generating a `WM_PAINT` message. This effectively means that repainting a window is a low-priority task which will not be done as long as the application is busy with other tasks. It also means that multiple invalidations are dealt with by a single `WM_PAINT` message, which is more efficient.

This is reasonable: as long as the user has not noticed that part of the window is invalid it is more important to deal with input. If the user notices and stops typing or mousing around, then the input queue will empty and there will be time to update the window.

Regardless of whether the window is synchronous or not, the result is shown in the attached figure. User actions cause `WM_COMMAND` messages, which update the internal representation (model), and then perform a `WinInvalidateRect()`, causing the system to generate a `WM_PAINT` and updating the window (view).

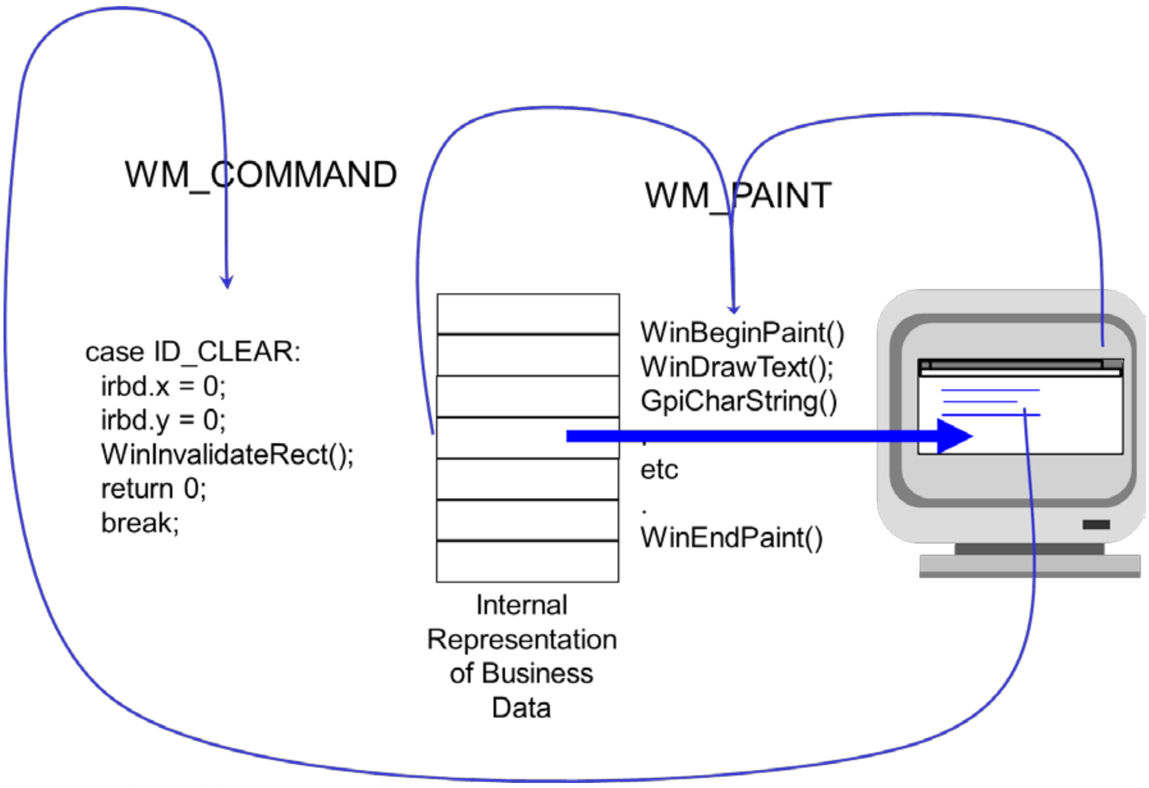


Figure 1. The Model-View-Controller approach to a PM program.

The beauty of this approach is that it allows easy extension into a multithreaded model. As mentioned above, presentation spaces are non-reentrant, so that two threads cannot write to the PS at the same time. But since there is only one place in this winproc where it writes to the presentation space, it is safe to extract this code and turn it into another thread.

So, in response to `WM_PAINT`, rather than working directly on the PS, which could take some time, the main thread would delegate another thread to do the repaint, and would immediately return from the winproc, so as to keep the system responsive (remember, because of PM's synchronous input queue model, the next message will not be read by any thread until this winproc returns, making the system non-responsive).

Now, the approach which many textbooks and courses take at this point is to have the `WM_PAINT` processing kick off another thread, using the `_beginthread()` or `DosCreateThread()` API's. However, there are two problems with this approach:

First, while threads provide a lightweight, cheap, form of multitasking when compared with spawning off another process, they are not free. It takes time to start another thread (allocation and initialisation of thread control blocks, allocation of a stack, and more), and this will cause a slight delay every time the window is to be drawn.

Secondly, what happens if two `WM_PAINT` messages are generated near-simultaneously, as might happen with a synchronous window? Now, two threads could be created and try to access the same ps simultaneously - precisely the situation we are trying to avoid.

Now, the latter situation can be avoided by using a mutex semaphore to protect the PS. However, we are trying to avoid this approach as well, although less strenuously.

A better technique would be to create a worker thread to handle the `WM_PAINT` processing before it is needed, either in the mainline of the program, or better still, in the `WM_CREATE` processing of the window. Before the thread is created, the creator code would have reset the event semaphore, so that when the thread starts running and calls `DosWaitEventSem()` it blocks immediately.

Now, when the `WM_PAINT` code runs, all it does is to post the event sem, which releases the painting thread to go off and paint in the PS, while the main thread returns. After the painting thread has finished with the PS, it resets the event semaphore and loops around to `DosWaitEventSem()` again.

Obviously, there's a little more to it than that, but you get the basic idea. The programmer must also provide a shared flag which the main thread can set to indicate that the painting thread should clean up and terminate; it's never a good idea to abruptly choke the life out of a thread with `DosKillThread()` - that API is for emergency use only.

The MVC approach to application and window implementation helps to keep things simple as the application increases in complexity.

Using Queues for Inter-Thread Communication

In the previous article, we discussed the MVC paradigm for development of PM programs and briefly discussed the possibility of allowing painting of a window to be done asynchronously, in a secondary thread. In this article, we shall examine the various techniques which can be used to set up a secondary thread to paint a window, with particular emphasis on mechanisms for communication between the threads.

The primary mechanism for inter-thread communication is one that is rarely mentioned in books on OS/2 programming, perhaps because it is so obvious. As a consequence, after immersion in OS/2 technical manuals, it is easy to overlook it as your mind reels with notions of pipes, queues, semaphores and the like. The mysterious technique? Simply sharing variables in memory!

Remember that processes own resources like memory, not threads. Consequently all the threads of a process enjoy equal access to memory, and can use variables to indicate state information.

There is therefore no fundamental objection to using variables in memory as flags or more sophisticated mechanisms for inter-thread synchronization or communication. However, there is one difficulty. Take, for example, a situation where two threads might attempt to print simultaneously (or paint in a window, come to that). We might decide to use a dword in memory as a 'printer in use' flag: if it is zero, then the printer is available, but once a thread starts printing, it should set the flag to -1 (FFFFFFFFH).

Now suppose that somehow, two threads decide to print near-simultaneously. The first thread executes some code which loads the value of the 'printer in use' flag into a processor register and then does a JZ (Jump on Zero) to its print routine. But before it can decrement the register to -1 and write it back, the system timeslices, and now a second thread does the same thing. It loads the flag into a register, does a JZ and then decrements it to -1, writes it back and starts to print. Several milliseconds later, the system timeslices again and eventually the first thread runs, writes back -1 and also starts to print, resulting in garbage output.

The basic problem is that the act of testing, and if clear, setting, a flag should be atomic and indivisible, i.e. it may not be interrupted part-way through. Now, in fact, we can arrange that under OS/2 by using critical sections, which are blocks of code which may not be interrupted by other threads in the same process. Calling `DosEnterCritSec()` causes other threads in this process to be suspended until a matching `DosExitCritSec()` API call is made. However, one should be aware that there are some restrictions on what can be done in a critical section, and that they are dangerous. Crashing while in a critical section can get really ugly.

For this reason, the OS provides a number of semaphore functions which provide the required functionality and more. Internally, they operate using critical sections in an extensively debugged and safe manner, and they also provide more generality.

Are variables in memory of any use, then? The answer is yes. A typical scenario is where the primary user interface thread of an application requires a thread to start a task, such as filling a list box, which might never be required to complete (the user might type the desired entry into the entry-field of a combo-box without needing to pick from the listbox component, for example).

Since the semaphore API's are organised to allow blocking of a thread until an event occurs or a resource becomes available, they are not appropriate here; rather we want the thread to continue running *until* an event occurs. A shared variable is ideal for this task. Simply call it `BOOL bcontinue`, set `bcontinue = TRUE` to start, and then if the background thread is to be stopped, set `bcontinue = FALSE`.

The background thread, of course, will loop around doing its stuff, using a construct of the form `while (bcontinue) { . . . }.` Following this, will be its termination and cleanup code.

This technique is also good for terminating general-purpose worker threads. While the system provides a `DosKillThread()` API, it should not be used, except as a last resort. DO NOT RELY ON `DosKillThread()` AS A WAY TO TERMINATE THREADS!! This API cannot always kill threads which are running at a low priority, or which are currently in K-mode, and it will not allow threads to perform termination and cleanup processing - for example to allow a thread started with `_beginthread()` to clean up the C run-time library. Instead, use a variable, as shown above.

Another structure which is useful for communicating with worker threads is the queue. OS/2 supports two different kinds of queues, and it is important not to get the two confused: PM message queues are created with `WinCreateMsgQueue()` and then read with `WinGetMsg()`, while the Control Program queues are create with `DosCreateQueue()` and read with `DosReadQueue()`.

We can certainly use PM message queues to communicate amongst threads, but remember the dreaded One-Tenth-Second Rule: if a thread creates a message queue, then it must return from each message within 1/10th second. If it can't do this, then the usual work-around is to create another thread, but how will we communicate with it? The way out of this circular circumstance is to create an Object Window, but that will have to wait for a later article.

Control program queues do not suffer from this restriction. They furthermore have some useful properties: they are fast, they can be used for interprocess communications (as can PM queues), they can be used in non-PM applications and subsystems, and they can be read in FIFO, LIFO, priority or arbitrary order.

One restriction on queues is that the only process which can read from a queue is its creator. This does not pose an architectural problem in inter-thread communication, however; it is perfectly permissible for several threads in a single process to read from a queue, in addition to the thread that originally created it.

In the following example, I have contrived an application which takes some time to paint a sine wave in its window, by the simple expedient of adding a loop:

```
for (j = 0; j < 100000; j++); // Slow down and suck up CPU cycles
```

between each pixel of the drawn sine wave. Depending upon the window size, it can take several seconds to repaint the window, during which time a single-threaded version of the program can noticeably make the system unresponsive - menus won't pull down, background windows can't be brought to the foreground and the system is to all intents and purposes dead for a few seconds.

The source code starts off by defining the usual stuff and including the usual header files. A preprocessor macro is used to support writing of a debug log for instructional purposes.

```
/* Queued multithreaded paint example */
#define DEBUG
/* We need to include symbolic constants, function prototypes,
structures and types for the system */
#define INCL_BASE
#define INCL_DOSQUEUES
#define INCL_WIN
#include <os2.h>

#include <stdio.h>
#include <malloc.h>
#include <math.h>

#include "\os290\lab15\pmassert.h"
```

Now we define the client window class and some messages which will allow the window-painting thread to 'talk back' to the main thread via its PM message queue:

```
/* A window class has a name, used in several places through the
source, so we define it here */
#define WC_HELLO "Hello"
#define UM_PAINTTITLE WM_USER
#define UM_NORMALTITLE WM_USER+1
```

Then come the usual forward reference for the winproc, and global variables (not a good idea, but hardly harmful in such a small program):

```
/* Forward reference for the window procedure */
MRESULT EXPENTRY SineWndProc(HWND hwnd, ULONG msg, MPARAM mp1, MPARAM
mp2);

/* We need variables for a frame window and a client window */
HWND hwndFrame, hwndClient;
HAB hab;            // Handle to an anchor block
```

In order to create a queue, we'll need a handle to refer to it (`DosCreateQueue()` will initialise this). We also need a structure which will be used to pass commands and parameters to the window-painting thread. For example, the handle of the window to be painted, the handle of the presentation space to paint in and the invalid rectangle are all obtained by the main thread when it calls `WinBeginPaint()` (which it must do in order not to be bombarded with more `WM_PAINT` messages). We also include a command field, so that we can tell the paint thread to either paint or to terminate itself (or do something else).

```
HQUEUE hq;

// Structure for inter-thread communication
typedef struct _PaintCommand {
    enum {T2PAINT, T2END } command;
    HWND hwndPaint;
    HPS hpsPaint;
    RECTL rclPaint;
} PAINTCOMMAND, *PPAINTCOMMAND;
```

The thread function must be correctly prototyped so that `DosCreateThread()` can be called. Note the use of `_System` linkage. We also need a thread ID variable.

```
/* Forward reference to thread function */
VOID _System PaintThread(ULONG ulParam);
TID tid;
```

The start of the main function is unremarkable, save for conditionally-compiled call to `freopen()` which allows us to use printf() calls to write to a log file.

```
int main(int argc, char *argv[], char *envp[])
{
    APIRET rc;           // Base OS return code
    HMQ hmq;             // Handle for a message queue
    QMSG qmsg;           // Queue message structure
    ULONG ulFrameFlags; // Frame creation flags - allow for frame
components
    PPAINTCOMMAND pcmd;

#ifdef DEBUG
    // Set up a debug log file
    freopen("LOG.TXT" , "w", stdout);
#endif
```

Now we need to create the queue. Note that the thread is created to be read in priority sequence. This means that we can make a T2END command 'leapfrog' over any pending T2PAINT commands, so that the user is not forced to watch the window repaint several times after he has chosen to quit. If there are multiple elements of equal priority in the queue, then they will be read in FIFO sequence anyway.

```
    /* Create the inter-thread communications queue */
    rc = DosCreateQueue(&hq, QUE_PRIORITY, "\\queues\\paint.que"); /*
Will read in FIFO order if equal priority */
    if (rc != 0) {
        printf("Failed to create queue, rc = %d\n", rc);
fflush(stdout);
    }
```

Next, we create the secondary thread:

```
    /* Create the secondary thread */
    rc = DosCreateThread(&tid, PaintThread, 0L, STACK_SPARSE, 80000);
#ifdef DEBUG
    if(rc) {
        printf("Failed to create secondary thread, rc = %d\n", rc);
fflush(stdout);
    }
#endif
```

From here, the code reads like a conventional PM program until we get to the termination code, where we have to kill the paint thread. We do this by allocating memory for a 'paint request' packet, and inserting the T2END command into it:

```
    /* Kill the paint thread */
    // Assemble a paint request packet
    pcmd = (PPAINTCOMMAND)malloc(sizeof(PAINTCOMMAND));
    pmassert(hab, pcmd);
    pcmd->hpsPaint = 0L;
    pcmd->hwndPaint = 0L;
    pcmd->command = T2END;
```

And then posting it to the queue:

```
    rc = DosWriteQueue(hq,
             0,
             sizeof(PAINTCOMMAND),
             (PVOID)pcmd,
             15);  // Priority higher than normal to preempt any
pending paints
```

Having done this, we must now wait until the paint thread terminates before destroying the window and destroying the queue:

```
    DosWaitThread(&tid, DCWW_WAIT);
    /* Destroying the frame automatically destroys all its children
*/
    WinDestroyWindow(hwndFrame);
    WinDestroyMsgQueue(hmq);
    WinTerminate(hab);
    rc = DosCloseQueue(hq);
    return 0;
}
```

Of course, as with most PM programs, the bulk of the work is done in the window procedure, which is mostly conventional:

```
MRESULT EXPENTRY SineWndProc(HWND hwnd, ULONG msg, MPARAM mp1, MPARAM
mp2)
{
    APIRET rc;
    PPAINTCOMMAND pcmd;
```

Until we come to the `WM_PAINT` processing, that is. Here, the first order of business is to assemble a paint request packed, first `malloc`'ing memory for it - note the consistent protocol that the sender `malloc`s, while the recipient thread `free`s.

```
        case WM_PAINT:
            // Assemble a paint request packet
            pcmd = (PPAINTCOMMAND)malloc(sizeof(PAINTCOMMAND));
            pmassert(hab, pcmd);
```

If we don't call `WinBeginPaint()` in order to clear the invalid rectangle before returning from the WM_PAINT message, the system will bombard us with `WM_PAINT` messages. So we can't just post off a packet to the paint thread and let it call `WinBeginPaint()` to get an hps and the invalid rectangle - we've got to do that here and pass those parameters in the packet:

```
            pcmd->hpsPaint = WinBeginPaint(hwnd, 0, &pcmd->rclPaint);
// Get a presentation space to paint in
            pcmd->hwndPaint = hwnd;
            pcmd->command = T2PAINT;
```

And now we write this packet to the queue:

```
            rc = DosWriteQueue(hq,
                    0,
                    sizeof(PAINTCOMMAND),
                    (PVOID)pcmd,
                    8);
```

While the paint thread is repainting the window, we want to signal that fact in the titlebar. So we arrange for the thread to post us a `UM_PAINTTITLE` message, and in response to this we change the titlebar text:

```
        case UM_PAINTTITLE:
            WinSetWindowText(WinWindowFromID(hwndFrame,FID_TITLEBAR),
"Eight Pi - Repainting");
            return (MRESULT) FALSE;
            break;
```

And do the reverse when the paint thread has finished:

```
        case UM_NORMALTITLE:
            WinSetWindowText(WinWindowFromID(WinQueryWindow(hwnd,
QW_PARENT), FID_TITLEBAR), "Eight Pi");
            return (MRESULT) FALSE;
            break;
```

The rest of the window procedure code is conventional. The paint thread starts off with the usual data definitions:

```
VOID _System PaintThread(ULONG ulParam)
{
    PPAINTCOMMAND pcmd;
    static ULONG ulCommandCounter = 1;
    HAB hab;
    REQUESTDATA request;
    ULONG uldatalength;
    BYTE ucPriority;
    RECTL rclWindow;
    APIRET rc;
    INT j;
    BOOL brc;
    POINTL ptl;
    LONG lScaleFactor, xaxis;
    double theta;
```

The first order of business is to call WinInitialize so as to obtain access to PM resources and then set the priority of the thread lower than a regular application thread, so that painting is not done at the expense of other system activities:

```
    hab = WinInitialize(0L);
    rc = DosSetPriority(PRTYS_THREAD, PRTYC_IDLETIME, 4, 0);
```

Then comes the loop which reads commands off the queue and processes them. First, we read a command:

```
    for(;;) {
        /* Read a command off the queue */
        rc = DosReadQueue(hq,            // Queue handle
                &request,                // request (not used)
                &uldatalength,            // length of queue element
(not used)
                (PPVOID)&pcmd,            // Passed data
                0,                       // Element code - read from
front of queue
                DCWW_WAIT,               // Wait (opp. DCWW_NOWAIT)
                &ucPriority,             // Priority (not used)
                0L);                     // Semaphore handle (not
used since synchronous)
```

And then, if it is a `T2END` command, we exit the loop, otherwise paint the window:

```
        if(pcmd->command == T2END) break;   // We're outa here
        WinQueryWindowRect(pcmd->hwndPaint, &rclWindow);
        // Set the title bar to show we're painting
        WinPostMsg(pcmd->hwndPaint, UM_PAINTTITLE, 0L, 0L);
        pmassert(hab, brc);
        WinFillRect(pcmd->hpsPaint, &pcmd->rclPaint, CLR_WHITE);
        lScaleFactor = (rclWindow.yTop - rclWindow.yBottom) / 2;
        xaxis = (LONG) rclWindow.yBottom + lScaleFactor;

        for(ptl.x = pcmd->rclPaint.xLeft-1; ptl.x < pcmd-
>rclPaint.xRight; ptl.x++) { // For every pixel column across the
invalid rect
            theta = 4 * (ptl.x - rclWindow.xLeft) * 2 * 3.141592654 /
(rclWindow.xRight - rclWindow.xLeft);
            ptl.y = xaxis + (LONG)( sin(theta)  * lScaleFactor );
           // if (ptl.y > pcmd->rclPaint.yTop || ptl.y < pcmd-
>rclPaint.yBottom) continue;
            GpiLine(pcmd->hpsPaint, &ptl);
            for (j = 0; j < 100000; j++); // Slow down and suck up
CPU cycles
        }
        brc = WinEndPaint(pcmd->hpsPaint);                   //
Release the presentation space
```

After painting the window, we let the main thread know we've finished, so it change the titlebar back, free the command packet memory, and then loop around to do it all over again:

```
        WinPostMsg(pcmd->hwndPaint, UM_NORMALTITLE, 0L, 0L);
        free(pcmd);
    } // Do it all over again
    DosExit(EXIT_THREAD, 0);
}
```

Here is the full program:

```
/* Queued multithreaded paint example */
#define DEBUG
/* We need to include symbolic constants, function prototypes,
structures and types for the system */
#define INCL_BASE
#define INCL_DOSQUEUES
#define INCL_WIN
#include <os2.h>

#include <stdio.h>
#include <malloc.h>
#include <math.h>

#include "\os290\lab15\pmassert.h"

/* A window class has a name, used in several places through the
source, so we define it here */
#define WC_HELLO "Hello"
#define UM_PAINTTITLE WM_USER
#define UM_NORMALTITLE WM_USER+1

/* Forward reference for the window procedure */
MRESULT EXPENTRY SineWndProc(HWND hwnd, ULONG msg, MPARAM mp1, MPARAM
mp2);

/* We need variables for a frame window and a client window */
HWND hwndFrame, hwndClient;
HAB hab;            // Handle to an anchor block

HQUEUE hq;

// Structure for inter-thread communication
typedef struct _PaintCommand {
    enum {T2PAINT, T2END } command;
    HWND hwndPaint;
    HPS hpsPaint;
    RECTL rclPaint;
} PAINTCOMMAND, *PPAINTCOMMAND;

/* Forward reference to thread function */
VOID _System PaintThread(ULONG ulParam);
TID tid;

int main(int argc, char *argv[], char *envp[])
{
    APIRET rc;          // Base OS return code
    HMQ hmq;            // Handle for a message queue
    QMSG qmsg;          // Queue message structure
    ULONG ulFrameFlags; // Frame creation flags - allow for frame
components
    PPAINTCOMMAND pcmd;

#ifdef DEBUG
    // Set up a debug log file
    freopen("LOG.TXT" , "w", stdout);
#endif

    /* Create the inter-thread communications queue */
    rc = DosCreateQueue(&hq, QUE_PRIORITY, "\\queues\\paint.que"); /*
Will read in FIFO order if equal priority */
    if (rc != 0) {
        printf("Failed to create queue, rc = %d\n", rc);
fflush(stdout);
    }

    /* Create the secondary thread */
    rc = DosCreateThread(&tid, PaintThread, 0L, STACK_SPARSE, 80000);
#ifdef DEBUG
    if(rc) {
        printf("Failed to create secondary thread, rc = %d\n", rc);
fflush(stdout);
    }
#endif

    /* Let PM know we're here, and it can allocate resources */
    hab = WinInitialize(0);

    /* Create a message queue to receive messages for our windows */
    hmq = WinCreateMsgQueue(hab, 0);

    /* Register a window class (winproc) to process messages for the
client window */
    if(!WinRegisterClass(hab,
                WC_HELLO,
                SineWndProc,
                CS_SYNCPAINT | CS_SIZEREDRAW,
                0L))
        DosExit(EXIT_PROCESS,1);

    /* Set up frame creation flags for a standard window, only no
menu, accelerator table or icon */
    ulFrameFlags = FCF_STANDARD & ~FCF_MENU & ~FCF_ACCELTABLE &
~FCF_ICON;

    /* Create a frame (and its children) and client windows */
    hwndFrame = WinCreateStdWindow(HWND_DESKTOP,    // Frame's parent
                    WS_VISIBLE,                     // Frame window
style
                    &ulFrameFlags,                  // Frame creation
flags
                    WC_HELLO,                       // Client window
class
                    "Eight Pi",                        // Titlebar
text
                    WS_VISIBLE,                     // Client window
style
                    (HMODULE)0L,                    // Resource
module handle - 0 for resources in .EXE
                    0L,                             // Resource ID
                    &hwndClient);                   // Variable to
receive client window handle

    if (hwndFrame == 0) {
        WinAlarm(HWND_DESKTOP, WA_ERROR);
        DosExit(EXIT_PROCESS, 1);
    }

    /* The 'event loop' reads messages and dispatches them until
WinGetMsg returns FALSE after reading WM_QUIT */
    while(WinGetMsg(hab, &qmsg, 0L, 0L, 0L))
        WinDispatchMsg(hab, &qmsg);
    /* Kill the paint thread */
    // Assemble a paint request packet
    pcmd = (PPAINTCOMMAND)malloc(sizeof(PAINTCOMMAND));
    pmassert(hab, pcmd);
    pcmd->hpsPaint = 0L;
    pcmd->hwndPaint = 0L;
    pcmd->command = T2END;
    // And post it to the queue
    rc = DosWriteQueue(hq,
             0,
             sizeof(PAINTCOMMAND),
             (PVOID)pcmd,
             15);  // Priority higher than normal to preempt any
pending paints
#ifdef DEBUG
    printf("T1: Sent T2END command to end T2\n"); fflush (stdout);
#endif
    DosWaitThread(&tid, DCWW_WAIT);
    /* Destroying the frame automatically destroys all its children
*/
    WinDestroyWindow(hwndFrame);
    WinDestroyMsgQueue(hmq);
    WinTerminate(hab);
    rc = DosCloseQueue(hq);
    return 0;
}

/* The 'guts' of the application is in the window procedure, which
implements behaviour of the client window */
MRESULT EXPENTRY SineWndProc(HWND hwnd, ULONG msg, MPARAM mp1, MPARAM
mp2)
{
    APIRET rc;
    PPAINTCOMMAND pcmd;
    static ULONG ulCommandCounter = 1;

    switch(msg) {
        case WM_CREATE:                     // Return false to allow
the system to
            return (MRESULT) FALSE;         // continue creating the
window
            break;                          // Return TRUE if there's
a problem

        case WM_ERASEBACKGROUND:
            return (MRESULT) FALSE;          // Return TRUE to allow
the frame to fill the client with white
            break;

        case WM_PAINT:                      // The real 'guts' of the
application are here
            // Assemble a paint request packet
            pcmd = (PPAINTCOMMAND)malloc(sizeof(PAINTCOMMAND));
            pmassert(hab, pcmd);
            pcmd->hpsPaint = WinBeginPaint(hwnd, 0, &pcmd->rclPaint);
// Get a presentation space to paint in
            pcmd->hwndPaint = hwnd;
            pcmd->command = T2PAINT;
            // And post it to the queue
            rc = DosWriteQueue(hq,
                    0,
                    sizeof(PAINTCOMMAND),
                    (PVOID)pcmd,
                    8);
#ifdef DEBUG
            printf("T1: Sent paint command %d to paint thread\n",
ulCommandCounter++); fflush(stdout);
#endif
            return (MRESULT) FALSE;             // Return FALSE to
indicate the message has been processed
            break;

        case UM_PAINTTITLE:
            WinSetWindowText(WinWindowFromID(hwndFrame,FID_TITLEBAR),
"Eight Pi - Repainting");
            return (MRESULT) FALSE;
            break;

        case UM_NORMALTITLE:
            WinSetWindowText(WinWindowFromID(WinQueryWindow(hwnd,
QW_PARENT), FID_TITLEBAR), "Eight Pi");
            return (MRESULT) FALSE;
            break;

        case WM_CLOSE:                          // Post back a
WM_QUIT to yourself
            WinPostMsg(hwnd, WM_QUIT, 0L, 0L);  // so that the event
loop terminates
            return (MRESULT) FALSE;
            break;

        default:        // Let the system handle all other messages
with default behaviour
            return WinDefWindowProc(hwnd, msg, mp1, mp2);
            break;
    }
    return (MRESULT) FALSE; // Shut the compiler up
}

/* Painting thread */

VOID _System PaintThread(ULONG ulParam)
{
    PPAINTCOMMAND pcmd;
    static ULONG ulCommandCounter = 1;
    HAB hab;
    REQUESTDATA request;
    ULONG uldatalength;
    BYTE ucPriority;
    RECTL rclWindow;
    APIRET rc;
    INT j;
    BOOL brc;
    POINTL ptl;
    LONG lScaleFactor, xaxis;
    double theta;

#ifdef DEBUG
    printf("T2: starting\n"); fflush(stdout);
#endif
    hab = WinInitialize(0L);
    rc = DosSetPriority(PRTYS_THREAD, PRTYC_IDLETIME, 4, 0);
```

```
#ifdef DEBUG
    printf("T2: Priority reduced, rc = %d\n", rc); fflush(stdout);
#endif
    for(;;) {
#ifdef DEBUG
        printf("T2: Entered loop\n"); fflush (stdout);
#endif
        /* Read a command off the queue */
        rc = DosReadQueue(hq,            // Queue handle
                &request,                // request (not used)
                &uldatalength,            // length of queue element
(not used)
                (PPVOID)&pcmd,            // Passed data
                0,                       // Element code - read from
front of queue
                DCWW_WAIT,               // Wait (opp. DCWW_NOWAIT)
                &ucPriority,             // Priority (not used)
                0L);                     // Semaphore handle (not
used since synchronous)
#ifdef DEBUG
        printf("T2: Command %d was read\n", ulCommandCounter++);
fflush(stdout);
#endif
        if(pcmd->command == T2END) break;   // We're outa here
        WinQueryWindowRect(pcmd->hwndPaint, &rclWindow);
        // Set the title bar to show we're painting
        WinPostMsg(pcmd->hwndPaint, UM_PAINTTITLE, 0L, 0L);
        pmassert(hab, brc);
        WinFillRect(pcmd->hpsPaint, &pcmd->rclPaint, CLR_WHITE);
        lScaleFactor = (rclWindow.yTop - rclWindow.yBottom) / 2;
        xaxis = (LONG) rclWindow.yBottom + lScaleFactor;

        for(ptl.x = pcmd->rclPaint.xLeft-1; ptl.x < pcmd-
>rclPaint.xRight; ptl.x++) { // For every pixel column across the
invalid rect
            theta = 4 * (ptl.x - rclWindow.xLeft) * 2 * 3.141592654 /
(rclWindow.xRight - rclWindow.xLeft);
            ptl.y = xaxis + (LONG)( sin(theta)  * lScaleFactor );
           // if (ptl.y > pcmd->rclPaint.yTop || ptl.y < pcmd-
>rclPaint.yBottom) continue;
            GpiLine(pcmd->hpsPaint, &ptl);
            for (j = 0; j < 100000; j++); // Slow down and suck up
CPU cycles
        }
        brc = WinEndPaint(pcmd->hpsPaint);                   //
Release the presentation space
#ifdef DEBUG
        printf("T2: WinEndPaint returned %d\n", brc); fflush(stdout);
#endif
        WinPostMsg(pcmd->hwndPaint, UM_NORMALTITLE, 0L, 0L);
        free(pcmd);
    } // Do it all over again
#ifdef DEBUG
    printf("T2: terminating\n"); fflush(stdout);
#endif
    DosExit(EXIT_THREAD, 0);
}
```

Notice, by the way, that unlike earlier examples in this series, which have simply repainted their entire client window area in response to WM_PAINT messages, this one only repaints the invalid rectangle, in the interest of performance.

This program is not a comprehensive exposition of queues, or multithreaded painting techniques, or the MVC paradigm, or of worker threads generally. But it does provide a basic introduction to some of the issues involved, as well as providing a small experimental platform. For example, what happens if the priority of the paint thread is not reduced?

Using Object Windows & Message Queues for MVC

In the previous section, we examined an implementation of the Model-View-Controller Paradigm for application design, using control program queues. We briefly mentioned the alternative use of Presentation Manager message queues, but gave no details.

Of course, the first problem that will occur to the perceptive reader is that message queues provide communication between windows, not threads or processes. While it should be possible to communicate with another process by getting its window handles, what about use in a simple application which simply has a single client window? Here, there is no window to which we can send or post messages.

However, the solution is at hand. And even better, it fits well with another set of modern programming concepts - those of object oriented programming.

In a previous section, we showed how a window can be used as the unit of encapsulation, through the use of window words. In this context, a window procedure can be thought of as a class (hence the `WinRegisterClass()` API), since it provides behaviour for all windows of the same type, and each window is an object, with its instance data stored in a structure pointed to by the window words.

It is tempting to investigate this further, by using a window to represent internal objects in a program. After all, windows can be created with zero size and position and not made visible, and we can then interact with them by sending or posting messages. However, there is a drawback: the system still sees such windows as at least potentially part of the user interface subsystem and sends them all kinds of messages which must be dealt with. In addition, such a window must respond to any messages almost immediately or it will hold up the reading of subsequent messages from the system's synchronous input queues.

Presentation Manager's designers obviously saw the potential for the use of the window concept for representation of internal objects and provided object windows for this purpose. An object window is descended from `HWND_OBJECT` (remember that conventional windows are descendants of `HWND_DESKTOP`) and is not part of PM's user interface subsystem. Consequently, object windows do not receive messages relating to user interface operation, and they need not process messages in less than 1/10th second.

In fact, the only system-defined messages which an object window will receive are `WM_CREATE`, which is used to create/allocate/open resources, `WM_DESTROY`, which is used for the converse operation, and `WM_QUIT`, which is used to terminate the thread running an object window.

An object window is generally run on its own thread, which is created using `DosCreateThread()`. The thread will need to have an anchor block, create an input queue, perhaps register a class for the object window and generally do the things that a main thread does:

```
    hab = WinInitialize(0L);
    hmq = WinCreateMsgQueue(hab, 0L);
```

In addition, it also needs to call `WinCancelShutdown()`. This API has two functions: when called with a `FALSE` parameter in response to a `WM_QUIT` message, it cancels the system shutdown which generated the `WM_QUIT` in the first place. This form is usually used in application primary UI threads. When called with the `TRUE` argument, it stops the system from sending `WM_QUIT` to this thread. Usually, the designer wants the system to send `WM_QUIT` to the primary thread only, which will then take care of terminating the other threads in a sequence and fashion which makes sense in the contect of the application. If all threads received `WM_QUIT` simultaneously, application termination would be difficult, to put it mildly:

```
    WinCancelShutdown(hmq, TRUE);        // We don't want to receive
WM_QUIT in event of system shutdown

    if(!WinRegisterClass(hab,
        WC_OPAINT,
        (PFNWP)wpPaintThread,
        0,
        sizeof(void *))) {
            DosBeep(440,1000);
            DosExit(EXIT_PROCESS,1);
          /* Couldn't register class */
    }
```

Finally, the object window is created. Note that it is created with `WinCreateWindow()`, rather than `WinCreateStdWindow()` and that it is a child of `HWND_OBJECT`:

```
    hwndObject = WinCreateWindow(HWND_OBJECT,     // hwndParent
                                 WC_OPAINT,    // Class
                                 NULL,         // Name
                                 0L,           // Style
                                 0,0,          // x,y
                                 0,0,          // cx, cy
                                 HWND_OBJECT,  // hwndOwner
                                 HWND_BOTTOM,  // hwndInsertBehind
                                 1,            // ID
                                 NULL,         // pCtlData
                                 NULL);        // pPresParams
```

At this point, an extra complication rears its ugly head: While the secondary thread was getting started and performing this initialisation, the main thread has been going about its business, creating its windows, as a result of this, the system will have generated a `WM_PAINT` message to make the application client window area visible. If this is handed off by the `WM_PAINT` processing in the main window posting a message to the object window, there is a much greater than vanishing chance that this message will be processed by the system before the object window has been created. As a result, the client window will not be repainted.

We need some mechanism to 'hold up' the main thread so that it does not proceed to create the application window(s) until such time as the object window is ready to service it. An event semaphore should do the job nicely. In the main thread, we set up the semaphore and create it in a reset state:

```
    /* Set up the wait semaphore to be initially reset */
    rc = DosCreateEventSem(NULL, &hevWait, 0L, FALSE);
```

then a little later, after registering the main window class, we wait until the secondary thread has created the object window:

```
    /* Got to wait until the secondary thread posts the wait
semaphore */
    rc = DosWaitEventSem(hevWait, -1L);
```

The corresponding code in the secondary thread posts the semaphore after returning from `WinCreateWindow()`:

```
    /* object window has been created so we can release the primary
thread */
    rc = DosPostEventSem(hevWait);
```

This takes care of that little synchronization issue.

Here is the complete program:

```
/* PM Queued multithreaded paint example */
#define DEBUG
/* We need to include symbolic constants, function prototypes,
structures and types for the system */
#define INCL_BASE
#define INCL_WIN
#include <os2.h>

#include <stdio.h>
#include <malloc.h>
#include <math.h>

#include "\prog21\lab15\pmassert.h"

/* A window class has a name, used in several places through the
source, so we define it here */
#define WC_HELLO "Hello"
#define WC_OPAINT "ObjectPaint"
#define UM_PAINTTITLE WM_USER
#define UM_NORMALTITLE WM_USER+1

#define T2PAINT WM_USER
#define T2END   WM_USER+1

/* Forward reference for the window procedure */
MRESULT EXPENTRY SineWndProc(HWND hwnd, ULONG msg, MPARAM mp1, MPARAM
mp2);
MRESULT EXPENTRY wpPaintThread(HWND hwnd, ULONG msg, MPARAM mp1,
MPARAM mp2);

/* We need variables for a frame window and a client window */
HWND hwndFrame, hwndClient, hwndObject = NULLHANDLE;

// Structure for inter-thread communication
typedef struct _PaintCommand {
    HWND hwndPaint;
    HPS hpsPaint;
    RECTL rclPaint;
} PAINTCOMMAND, *PPAINTCOMMAND;

// Event semaphore for thread synchronization
HEV hevWait;

/* Forward reference to thread function */
VOID _System PaintThread(ULONG ulParam);
TID tid;

int main(int argc, char *argv[], char *envp[])
{
    HAB hab;            // Handle to an anchor block
    APIRET rc;          // Base OS return code
    HMQ hmq;            // Handle for a message queue
    QMSG qmsg;          // Queue message structure
    ULONG ulFrameFlags; // Frame creation flags - allow for frame
components
    PPAINTCOMMAND pcmd;

#ifdef DEBUG
    // Set up a debug log file
    freopen("LOG.TXT" , "w", stdout);
#endif
```

```
    /* Set up the wait semaphore to be initially reset */
    rc = DosCreateEventSem(NULL, &hevWait, 0L, FALSE);

    /* Create the secondary thread */
    rc = DosCreateThread(&tid, PaintThread, 0L, STACK_SPARSE, 80000);
#ifdef DEBUG
    if(rc) {
        printf("Failed to create secondary thread, rc = %d\n", rc);
fflush(stdout);
    }
#endif

    /* Let PM know we're here, and it can allocate resources */
    hab = WinInitialize(0);

    /* Create a message queue to receive messages for our windows */
    hmq = WinCreateMsgQueue(hab, 0);

    /* Register a window class (winproc) to process messages for the
client window */
    if(!WinRegisterClass(hab,
                WC_HELLO,
                SineWndProc,
                CS_SYNCPAINT | CS_SIZEREDRAW,
                0L))
        DosExit(EXIT_PROCESS,1);

    /* Got to wait until the secondary thread posts the wait
semaphore */
    rc = DosWaitEventSem(hevWait, -1L);

    /* Set up frame creation flags for a standard window, only no
menu, accelerator table or icon */
    ulFrameFlags = FCF_STANDARD & ~FCF_MENU & ~FCF_ACCELTABLE &
~FCF_ICON;

    /* Create a frame (and its children) and client windows */
    hwndFrame = WinCreateStdWindow(HWND_DESKTOP,    // Frame's parent
                    WS_VISIBLE,                     // Frame window
style
                    &ulFrameFlags,                  // Frame creation
flags
                    WC_HELLO,                       // Client window
class
                    "Eight Pi",                        // Titlebar
text
                    WS_VISIBLE,                     // Client window
style
                    (HMODULE)0L,                    // Resource
module handle - 0 for resources in .EXE
                    0L,                             // Resource ID
                    &hwndClient);                   // Variable to
receive client window handle

    if (hwndFrame == 0) {
        WinAlarm(HWND_DESKTOP, WA_ERROR);
        DosExit(EXIT_PROCESS, 1);
    }
    /* The 'event loop' reads messages and dispatches them until
WinGetMsg returns FALSE after reading WM_QUIT */
    while(WinGetMsg(hab, &qmsg, 0L, 0L, 0L))
        WinDispatchMsg(hab, &qmsg);

    /* Kill the paint thread */
    WinPostMsg(hwndObject, WM_QUIT, 0L, 0L);
#ifdef DEBUG
    printf("T1: Posted WM_QUIT message to end T2\n"); fflush
(stdout);
#endif
    DosWaitThread(&tid, DCWW_WAIT);
    /* Destroying the frame automatically destroys all its children
*/
    WinDestroyWindow(hwndFrame);
    WinDestroyMsgQueue(hmq);
    WinTerminate(hab);
    return 0;
}

/* The 'guts' of the application is in the window procedure, which
implements behaviour of the client window */
MRESULT EXPENTRY SineWndProc(HWND hwnd, ULONG msg, MPARAM mp1, MPARAM
mp2)
{
    APIRET rc;
    PPAINTCOMMAND pcmd;
    static ULONG ulCommandCounter = 1;

    switch(msg) {
        case WM_CREATE:                      // Return false to allow
the system to
            return (MRESULT) FALSE;          // continue creating the
window
            break;                           // Return TRUE if there's
a problem

        case WM_ERASEBACKGROUND:
            return (MRESULT) FALSE;           // Return TRUE to allow
the frame to fill the client with white
            break;

        case WM_PAINT:                       // The real 'guts' of the
application are here
            // Assemble a paint request packet
            pcmd = (PPAINTCOMMAND)malloc(sizeof(PAINTCOMMAND));
            pmassert(WinQueryAnchorBlock(hwnd), pcmd);
            pcmd->hpsPaint = WinBeginPaint(hwnd, 0, &pcmd->rclPaint);
// Get a presentation space to paint in
            pcmd->hwndPaint = hwnd;
            // And post it to the object window
            rc = WinPostMsg(hwndObject,
                    T2PAINT,
                    MPFROMP(pcmd),
                    0L);
#ifdef DEBUG
            printf("T1: Sent paint command %d to paint thread using
hwnd %08x\n", ulCommandCounter++, hwndObject); fflush(stdout);
#endif
            return (MRESULT) FALSE;              // Return FALSE to
indicate the message has been processed
```

```
            break;

        case UM_PAINTTITLE:
            WinSetWindowText(WinWindowFromID(hwndFrame,FID_TITLEBAR),
"Eight Pi - Repainting");
            return (MRESULT) FALSE;
            break;

        case UM_NORMALTITLE:
            WinSetWindowText(WinWindowFromID(WinQueryWindow(hwnd,
QW_PARENT), FID_TITLEBAR), "Eight Pi");
            return (MRESULT) FALSE;
            break;

        case WM_CLOSE:                          // Post back a
WM_QUIT to yourself
            WinPostMsg(hwnd, WM_QUIT, 0L, 0L);  // so that the event
loop terminates
            return (MRESULT) FALSE;
            break;

        default:        // Let the system handle all other messages
with default behaviour
            return WinDefWindowProc(hwnd, msg, mp1, mp2);
            break;
    }
    return (MRESULT) FALSE; // Shut the compiler up
}

/* Painting thread */

VOID _System PaintThread(ULONG ulParam)
{
    HAB hab;            // Handle to an anchor block
    HMQ hmq;
    QMSG qmsg;
    APIRET rc;

#ifdef DEBUG
    printf("T2: starting\n"); fflush(stdout);
#endif
    rc = DosSetPriority(PRTYS_THREAD, PRTYC_IDLETIME, 4, 0);
#ifdef DEBUG
    printf("T2: Priority reduced, rc = %d\n", rc); fflush(stdout);
#endif

    hab = WinInitialize(0L);
    hmq = WinCreateMsgQueue(hab, 0L);

    WinCancelShutdown(hmq, TRUE);        // We don't want to receive
WM_QUIT in event of system shutdown

    if(!WinRegisterClass(hab,
        WC_OPAINT,
        (PFNWP)wpPaintThread,
        0,
        sizeof(void *))) {
            DosBeep(440,1000);
            DosExit(EXIT_PROCESS,1);
           /* Couldn't register class */
    }
    hwndObject = WinCreateWindow(HWND_OBJECT,      // hwndParent
                                    WC_OPAINT,     // Class
                                    NULL,          // Name
                                    0L,            // Style
                                    0,0,           // x,y
                                    0,0,           // cx, cy
                                    HWND_OBJECT,   // hwndOwner
                                    HWND_BOTTOM,   // hwndInsertBehind
                                    1,             // ID
                                    NULL,          // pCtlData
                                    NULL);         // pPresParams

    pmassert(hab,hwndObject);

    /* object window has been created so we can release the primary
thread */
    rc = DosPostEventSem(hevWait);

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

    WinDestroyWindow(hwndObject);
    WinTerminate(hab);
#ifdef DEBUG
    printf("T2: terminating\n"); fflush(stdout);
#endif
    DosExit(EXIT_THREAD, 0);
}

MRESULT EXPENTRY wpPaintThread(HWND hwnd, ULONG msg, MPARAM mp1,
MPARAM mp2)
{
    PPAINTCOMMAND pcmd;
    static ULONG ulCommandCounter = 1;
    RECTL rclWindow;
    INT j;
    BOOL brc;
    POINTL ptl;
    LONG lScaleFactor, xaxis;
    double theta;

    switch(msg) {
        case WM_CREATE:
            return (MRESULT) 0L;
            break;

        case WM_DESTROY:
            return (MRESULT) 0L;
            break;

        case T2PAINT:
            pcmd = (PPAINTCOMMAND)PVOIDFROMMP(mp1);
            WinQueryWindowRect(pcmd->hwndPaint, &rclWindow);
            // Set the title bar to show we're painting
            WinPostMsg(pcmd->hwndPaint, UM_PAINTTITLE, 0L, 0L);
            WinFillRect(pcmd->hpsPaint, &pcmd->rclPaint, CLR_WHITE);
            lScaleFactor = (rclWindow.yTop - rclWindow.yBottom) / 2;
            xaxis = (LONG) rclWindow.yBottom + lScaleFactor;
            for(ptl.x = pcmd->rclPaint.xLeft-1; ptl.x < pcmd-
>rclPaint.xRight; ptl.x++) { // For every pixel column across the
invalid rect
                theta = 4 * (ptl.x - rclWindow.xLeft) * 2 *
3.141592654 / (rclWindow.xRight - rclWindow.xLeft);
                ptl.y = xaxis + (LONG)( sin(theta)  * lScaleFactor );
                GpiLine(pcmd->hpsPaint, &ptl);
                for (j = 0; j < 100000; j++); // Slow down and suck
up CPU cycles
            }
#ifdef DEBUG
            brc = WinEndPaint(pcmd->hpsPaint);                   //
Release the presentation space
            printf("T2: WinEndPaint returned %d\n", brc);
fflush(stdout);
#endif
            WinPostMsg(pcmd->hwndPaint, UM_NORMALTITLE, 0L, 0L);
            free(pcmd);
            return (MRESULT) FALSE;
            break;

        default:
            return WinDefWindowProc(hwnd, msg, mp1, mp2);
            break;
    } //end switch(msg)
    return 0L;
}
```

Subclassing

Subclassing is an object-oriented concept, and is possible in Presentation Manager because it is an object-orientedsystem, in certain senses. An object is a combination of instance data and methods, while a window can be viewed as the same thing: the instance data is stored in window words, while the methods are implemented in the window procedure.

If, in an object-oriented system, one can subclass an existing class which almost provides and desired behavior and modify its methods until it is exactly right, can one not do the same thing with window classes? The answer is yes: you can over-ride the behaviour of a window by intercepting messages to it and processing them in your own window procedure or allowing them to pass on to the original winproc. Notice that this is actually subclassing a window, rather than a class. Only that window will be affected, and it cannot be subclassed until after it has been created, so that the `WM_CREATE` message processing cannot be intercepted. Nonetheless, this is a surprisingly useful technique.

Subclassing can be used to modify the behaviour of existing classes. For example, originally entryfield controls always displayed what was typed into them, and so a common application for subclassing was the interception of `WM_CHAR` messages to catch passwords while displaying asterisks (today, one might use `ES_UNREADABLE` style on the entryfield, which does the same thing). However, entry-fields are still often subclassed to support editing of input, or addition of pop-up menus for cut/copy/paste and other useful features.

Here's how it is done: the `WinSubclassWindow()` API will allow the programmer to specify a new address for the window procedure for a window, and returns the address of the old window procedure. For example:

```
        case WM_INITDLG:
            /* Subclass the ID_SRCH_ABBREV entry field */
            OldWndProc = WinSubclassWindow(WinWindowFromID(hwnd,
ID_SRCH_ABBREV), UCEFWndProc);
```

In this case, `OldWndProc` is a static variable defined to be of type `PFNWP` (pointer to function which is a window procedure). Because the entryfield to be subclassed is in a dialog window, we know its ID but not its handle, so we use WinWindowFromID to get the handle. And UCEFWndProc is the Upper Case Entry Field winproc:

```
MRESULT EXPENTRY UCEFWndProc(HWND hwnd, ULONG msg, MPARAM mp1, MPARAM
mp2)
{
    switch(msg) {
        case WM_CHAR:
            mp2 =
MPFROM2SHORT(toupper(SHORT1FROMMP(mp2)),SHORT2FROMMP(mp2));
        default:
            return OldWndProc(hwnd, msg, mp1, mp2);
    }
    return (MRESULT) FALSE;
}
```

This window procedure simply intercepts all `WM_CHAR` messages, and converts short 1 of mp2 to upper case. It then calls OldWndProc(), as for all other messages. For intercepted messges, a subclassing window procedure can perform some pre- or post-processing around the base window procedure, as shown here, or can completely implement the desired behaviour, without calling the base winproc at all. Of course, if you don't call the base window procedure ever, for any message, there was no point in subclassing as you have completely implemented the window class.

The example code shown here can be seen in a modified version of Lab 10 called LAB10A.C, in the LAB10 subdirectory.

Dialog Windows

Dialog windows are windows which are usually displayed for short-term user interaction. They are typically used for font selection, data entry, file selection and similar interactions.

In most regards, dialog windows behave very much like conventional application windows, but there are some important distinctions. A dialog window has a window procedure that is structurally similar to that for a conventional window, but it does not have any class and `WinRegisterClass()` is not called for it.

A dialog window is actually a subclassed frame window. This has several interesting implications. The first is that your window procedure code for the dialog will never see the `WM_CREATE` message, since the frame window receives `WM_CREATE` before being subclassed. Instead, the dialog is sent a `WM_INITDLG` message, and this can be used in exactly the same way, to initialize resources required by the dialog.

The second implication is that, since it is a subclassing window procedure, your dialog window procedure should not call `WinDefWindowProc()`. Instead, it needs to call the original frame window procedure to perform default processing, and it should do this by calling `WinDefDlgProc()`.

Thirdly - and this is more subtle - you should remember that your code is running in the winproc for the frame itself, and not a client window. There is no frame/client relationship here: all the children of the frame are control windows like entry fields, list boxes and pushbuttons. If you want to, for example, display a bitmap background in your dialog, remember that there is no client.

The reason for subclassing the frame window is to provide the basic window-handling logic and add functionality such as processing of the tab and arrow keys for movement between control groups, selection of radio buttons, etc.

Conventions

A dialog window is typically used to obtain information from the user - a filename, font name and size, or perhaps a user ID and password. However, there is no way that a dialog, run by an API such as `WinDlgBox()`, can return several pieces of information. The `WinDlgBox()` API is of type ULONG, and the convention that has been adopted is that the API should return the window ID of the button which was used to dismiss the dialog - typically `DID_OK` or `DID_CANCEL`. In the dialog procedure, this is passed as an argument to `WinDismissDlg()`, and becomes the returned value of the `WinDlgBox()` API.

So now we have a way to determine whether the dialog ran successfully or not, but what about the data the user entered? The usual way of dealing with this is to pass a structure to the dialog window, which will be filled in before it is dismissed. A pointer to the structure is passed as one of the parameters of the `WinDlgBox()` call, and is received in the dialog procedure as mp2 of the `WM_INITDLG` message (mp1 is the handle of the control window which is to receive focus). *Important*: the first member of the passed structure must be a USHORT which is initialized to the size of the structure. This is required for thunking purposes.

Here is a simple example:

```
                case ID_SETCOLOR: {
                    BGColor.cbSize = sizeof(BGColor);
                    BGColor.red = (BYTE)(ulBGColor >> 16 &
0x000000ff);
                    BGColor.green = (BYTE)(ulBGColor >> 8 &
0x000000ff);
                    BGColor.blue = (BYTE)(ulBGColor & 0x000000ff);
                    if (WinDlgBox(HWND_DESKTOP,   /* Parent window -
the desktop */
                                  hwnd,           /* Owner window -
this window */
                                  ColorDlg,       /* The dialog
procedure */
                                  0,              /* Module handle -
the EXE */
                                  ID_DLGCOLOR,    /* Dialog resource
ID */
                                  &BGColor) == DID_OK) {
                        ulBGColor = 65536 * BGColor.red + 256 *
BGColor.green + BGColor.blue;
                        WinInvalidateRect(hwnd, NULL, FALSE);
                    }
                }
                break;
```

Note that the structure being passed to the dialog starts with a `USHORT` called `cbSize`, which is initialized to the size of the structure. The parent window of the dialog is usually the desktop, not the application client window. Why? The dialog will be clipped to its parent, and if the application client is fairly small, it is possible that it might obscure the OK and Cancel buttons on the dialog, making it impossible to dismiss. If the dialog is modal, this may make it impossible to quit the application. Messy.

The dialog window itself is created from a dialog template which is a resource attached to the .EXE file. We'll see how to create that later. If the `WinDlgBox()` API returns `DID_OK`, then the user clicked on the OK button (or otherwise dismissed the dialog, e.g. by double-clicking on a list-box item) and we go ahead and use the (presumably modified) values in the structure to update the application window.

Creating a Dialog Window

Dialog windows are usually designed using the Dialog Editor, which is one of the components of the OS/2 Developer Toolkit. Using the Dialog Editor is real kindergarten stuff: you just click on the window controls you want in a palette, and then click where you want to place them on the dialog.

Notice that each control should be given an identifier. The dialog editor will automatically assign an integer identifier to each control, but you should augment this by entering symbolic ID's. When the dialog is saved, the editor will create a source file - filename.DLG - a resource file - filename.RES - and a include file - filename.H which contains the symbolic values. This should then be included into the source file for the dialog procedure and possibly the client program.

The dialog editor has facilities for aligning controls, laying them out on a grid pattern and generally getting the visual appearance right.

Generally, the application will have other resources, in the form of a .RC resource script file which will have to be bound to the application .EXE file. If left to its own devices, the makemake utility will generate a make file which has two resource bind steps, but this will not work, as the second resource bind overwrites the resources from the first. Only one .RES file can be bound to an .EXE file, so this approach cannot work.

Instead the main .RC file should be edited to #include the dialog header (.H) file, and also to rcinclude the .DLG file. The .RES file containing the dialogs should be ignored during the make process, and only used to reload the dialogs into the dialog editor.

The Dialog Procedure

The dialog procedure is very like a conventional window procedure, with the differences noted above: default call to `WinDefDlgProc()` rather than `WinDefWindowProc()`; no `WM_CREATE` message, instead `WM_INITDLG`; and no client window area - the frame paints a grey background by default.

Modality

Most application dialogs are modal; that is, they are created using the `WinDlgBox()` API, and while they are displayed, input to other application windows is disabled. This is because `WinDlgBox()` incorporates its own event loop (while (WinGetMsg()) WinDispatchMsg();) which filters input and dispatches only messages for the dialog window and its children

The `WinLoadDlg()` API produces a modeless dialog box which will remain on the screen while the user works with it and other application windows. It can be made modal by calling the `WinProcessDlg()` API. However, modeless dialogs are generally more convenient for the user: witness the search and replace dialog in EPM, which allows the user to perform other editing tasks while searching for a string. Modeless dialogs should be used wherever possible.

The `WinCreateDlg()` API creates a modeless dialog from a dialog template in memory, not from the module resources. This can be useful when you want to dynamically build a dialog in memory. However, in the author's experience in building a generic SQL database front end, it is easier to build the on-screen form from WinCreateWindow() function calls than to build the dialog resource structures in memory.

Standard Dialogs

The system provides two standard dialogs for the commonest requirements: file open/save and font selection. The latter dialog, in particular, can save a lot of work.

In the bad old days of PM programming, every programmer used to write his own file open and save dialogs, and they all looked different. The DeScribe word processor, for example, had a dialog with three list boxes: one for drive selection, one for subdirectory and one for file, and this was quite different from every other application. The inconsistency annoys users and increases training and support costs (though not by much, admittedly). Plus, it costs programmers lots of time and money.

In the Microsoft OS/2 1.x SDK was a sample program showing how to encapsulate a dialog in a dynamic link library, and by happy coincidence this DLL just happened to implement a file open / save dialog. It wasn't long before programmers were using this DLL and shipping it with their applications. In OS/2 2.0, IBM standardised the DLL and added functionality.

The two standard dialogs are driven by structures which should be initialized before calling the corresponding DLL API (the DLL's have a single entry point). Although the structures look complex, they can basically be used by initializing almost everything to zeros and just calling the API. Only the most sophisticated applications would require extensive use of the structure members.

For the file open/save dialog, the structure looks like this:

```
cbSize;           /* Size of FILEDLG structure.         */
fl;               /* FDS_ flags. Alter behavior of dlg. */
ulUser;           /* User defined field.                */
lReturn;          /* Result code from dialog dismissal. */
lSRC;             /* System return code.                */
pszTitle;         /* String to display in title bar.    */
pszOKButton;      /* String to display in OK button.    */
pfnDlgProc;       /* Entry point to custom dialog proc. */
pszIType;         /* initial EA type filter. */
papszITypeList;   /* Type strings.  */
pszIDrive;        /* Initial drive.     */
papszIDriveList;  /* Drive strings.  */
hMod;             /* Custom File Dialog template.       */
szFullFile[CCHMAXPATH]; /* Initial  path and file.      */
papszFQFilename;  /* FQFname*/
ulFQFCount;       /* Number of files selected           */
usDlgId;          /* Custom dialog id.                  */
x;                /* X coordinate of the dialog         */
sEAType;          /* Selected file's EA Type.           */
```

The most important members to initialize are the `cbSize` field and the `fl` flags (especially `FDS_OPEN_DIALOG` or `FDS_SAVEAS_DIALOG`). Everything else can be zero and it will still work.

Here's the logic:

```
FILEDLG fildlg;
HFILE hFile;
memset(&fildlg, NULL, sizeof(FILDLG));
fildlg.cbsize=sizeof(FILDLG);
fildlg.fl = FDS_OPEN_DIALOG | FDS_CENTER;
fildlg.pszTitle = "Open Edit File";
hwndFileDlg = WinFileDlg(HWND_DESKTOP, hwndOwner, &fildlg);
if(hwndFileDlg && (fildlg.lReturn == DID_OK))
     rc = DosOpen(fild.szFullFile, &hFile, &ulAction, . . .);
```

The trick is to use `memset()` to zero out the structure memory and then just set what is required. In this case, the dialog is to be a Open dialog, and I've also set the title of the dialog rather than using the default. The `WinFileDlg()` API takes three arguments: the parent window (usually the desktop, for reasons previously discussed), the owner window and the address of the `FILEDLG` structure. Notice that we check for the API returning a valid dialog window handle and also that the `lReturn` field is set to `DID_OK` so that we know the dialog was displayed correctly and the user didn't press cancel. The filename is returned as a fully qualified pathname in the `szFullFile` member.

The font selection dialog structure is much more impressive; some might say alarming:

```
cbSize;        sizeof(FONTDLG)
hpsScreen;          Screen presentation space
hpsPrinter;         Printer presentation space
pszTitle;      Application supplied title
pszPreview;         String to print in preview wndw
pszPtSizeList;Application provided size list
pfnDlgProc;         Dialog subclass procedure
pszFamilyname;Family name of font
fxPointSize;   Point size the user selected
fl;            FNTS_* flags - dialog styles
flFlags;       FNTF_* state flags
flType;        Font type option bits
flTypeMask;         Mask of which font types to use
flStyle;       The selected style bits
flStyleMask;   Mask of which style bits to use
clrFore;       Selected foreground color
clrBack;       Selected background color
ulUser         Blank field for application
lReturn;       Return Value of the Dialog
lSRC;               System return code.
lEmHeight;          Em height of the current font
lXHeight;      X height of the current font
lExternalLeading;  External Leading of font
hMod;               Module to load custom template
fAttrs;        Font attribute structure
sNominalPointSize; Nominal Point Size of font
usWeight;      The boldness of the font
usWidth;       The width of the font
x;             X coordinate of the dialog
y;             Y coordinate of the dialog
usDlgId;       ID of a custom dialog template
usFamilyBufLen;     Length of family buffer provided
usReserved;         reserved
```

However, as before, the secret is to ignore most of the fields for simple applications. In fact, most of the fields relate to typographical font handling considerations beyond the scope of this course, although they are dealt with in course OS291, Advanced Presentation Manager and OS/2 Programming.

The most important fields to initialize are the cbSize field, the fl flags field, the

Once the WinFontDlg() function has returned, the fattrs structure will have been initialized and can be used to create a logical font by calling GpiCreateLogFont().

GPI

The Graphics Programming Interface is the major subsystem of Presentation Manager concerned with graphics. The GPI combines the benefits of a high-level graphics programming language with an intelligent device interface.

The high-level graphics programming language features:

- Stored Picture Elements (it stores graphic orders, a kind of interpreted graphics language)
- Hierarchical Picture Construction (segments allow graphics to be composed of 'subroutines')
- Picture Editing / Replacement (elements can be inserted and deleted as required)
- Input / Picture Correlation (automatically calculates which elements need to be repainted, or which are being selected by the mouse pointer)
- Automatic Picture Repair (needn't even repaint - the GPI can save the window contents in some circumstances)

The device interface:

- Lets the device do what it can
- Makes all devices look the same
- Provides information on the interface

Historically, the GPI is rooted in some older IBM mainframe software, such as GDDM (Graphical Data Display Manager) and the 3270 Graphics Control Program, coupled with newer standards such as PHIGS (Programmers Hierarchical Graphics Standard) and ANSI GKS (Graphical Kernel Standard). (PM itself is modelled on Smalltalk and the Microsoft Windows GDI, as well as the Computer Graphics Interface).

The GPI is different from earlier standards in two ways: It was the first system to combine support for both raster and vector devices, and the first to allow device sharing - serial sharing of printers through a complex spooler subsystem and concurrent use of interactive display via overlapping windows.

Basic GPI Concepts

There are some basic concepts to get under your belt before tackling the GPI. The GPI is the most complex component of OS/2, with hundreds of API's, but fortunately they fall into place once you have the basics under control.

A Device Context is the means of writing data to an output device. It is both the device driver and the physical device itself (if any). There are several types of device context:

- Screen device context
- Memory device context
- Metafile device context
- Other device device context (printer, plotter etc)

A specialised subset of the other device device context called an information device contexts allows querying, for example, to find out the default paper size when an application first creates a new document, or to query supported fonts.

The device context is associated with a Presentation Space. Drawing into the PS causes output to the associated DC. Since a PS is a 'logical device' which is completely generic, it cannot support device-specific functions like spooler job management, and so a `DevEscape()` function is provided which allows direct output.

To open a device context, use `DevOpenDC()` or `WinOpenWindowDC()`. The `DevOpenDC()` API supports various attributes:

- OD_QUEUED
- OD_DIRECT
- OD_INFO
- OD_METAFILE
- OD_MEMORY
- OD_METAFILE_NOQUERY

A Presentation Space is like a scratchpad or palette which the GPI uses to assemble the completed image. It is modal in operation, and consists of the following:

- Segment store
- Definition of symbol sets and fonts
- Definition of line-type sets
- Various controls, e.g. draw controls
- Logical color table / color palette
- Viewing pipeline, down to and including the page and page window

The Presentation Space is the key to using the GPI. It is a generalization of the device context of the GPI. There are three types of PS:

- Normal - full state preserved over time, supports segments, retained graphics, etc. and can be associated with any device context for true WYSIWYG operation.
- Micro - state preserved, but no retained graphics. Useable with mutiple device contexts, though.
- Micro-cached - usable with one DC only, typically screen. You must release the cached-micro PS before returning from processing a message, as there are only sixteen of these in PM and it uses them heavily for drawing controls

OS/2 1.x actually supports one other type of PS, which OS/2 2,x and Warp continue to support for 16-bit applications only: the Advanced VIO (AVIO) PS. The idea is that a character mode DOS application can quickly be ported to a VIO OS/2 application, since the OS/2 Vio subsystem API's map from the ROM BIOS INT 10H functions. However, such an application does not have any access to PM functionality such as resizable windows, menu bar, dialog windows and so on. The AVIO PS provides a PS in which character cell coordinates are used and allows the application to be ported to an intermediate stage before complete rewriting for PM/GPI. In effect, the application runs in a resizeable window with black characters on a white background, rather than fixed 80 x 25 white text on black background.

GPI Primitives

Drawing into a presentation space is accomplished by calling the GPI Primitive API's. Each primitive will perform some drawing function or store graphics elements into a segment (see below), depending on the current drawing mode.

To use the primitives, remember that GPI is a modal system. Each primitive will operate at or from the current position, in the current width, color, etc. Use `GpiMove()` or `GpiSetCurrentPos()` to position before starting to draw.

Primitives Quick Reference

Characters

`GpiCharString()`

`GpiCharStringAt()`

`GpiCharStringPos()`

`GpiCharStringPosAt()`

Lines

`GpiLine()`

`GpiPolyLine()`

`GpiBox()`

Arcs

`GpiFullArc()`

`GpiPartialArc()`

`GpiPointArc()`

`GpiPolyFillet()`

`GpiPolyFilletSharp()`

`GpiPolySpline()`

Markers

`GpiMarker()`

`GpiPolyMarker()`

Areas

`GpiBeginArea()`

`GpiEndArea()`

`GpiBeginPath()`

`GpiEndPath()`

`GpiFillPath()`

`GpiStrokePath()`

`GpiOutlinePath()`

`GpiModifyPath()`

`GpiCloseFigure()`

Images

`GpiImage()`

`GpiLoadBitmap()`

`GpiCreateBitMap()`

`GpiDrawBits()`

`WinDrawBitmap()`

`GpiBitBlt()`

`GpiWCBitBlt()`

Attributes on Primitives

There are two fundamentally different ways to modify the current attributes, thereby affecting all subsequent drawing primitives. The first is to use a BUNDLE structure to set or default a number of attributes simultaneously, while the second is to use specific calls to alter attributes individually.

The various primitives are broken into five types:

•	Area primitives	`AREABUNDLE`
•	Character primitives	`CHARBUNDLE`
•	Image primitives	`IMAGEBUNDLE`
•	Line primitives	`LINEBUNDLE`
•	Marker primitives	`MARKERBUNDLE`

Each type has a corresponding bundle structure which allows a number of attributes to be set simultaneously. For example, the line-attributes `LINEBUNDLE` structure looks like this:

```
typedef struct _LINEBUNDLE {
LONG     lColor;         /*  Line foreground color. */
LONG     lBackColor;     /*  Line background color. */
USHORT   usMixMode;      /*  Line foreground-mix mode. */
USHORT   usBackMixMode;  /*  Line background-mix mode. */
FIXED    fxWidth;        /*  Line width. */
LONG     lGeomWidth;     /*  Geometric line width. */
USHORT   usType;         /*  Line type. */
USHORT   usEnd;          /*  Line end. */
USHORT   usJoin;         /*  Line join. */
USHORT   usReserved;     /*  Reserved. */
 } LINEBUNDLE;

typedef LINEBUNDLE *PLINEBUNDLE;
```

Each field specifies some attribute used in line drawing (this covers lines, arcs, splines, ellipses, boxes and all the other shapes drawn with lines). In addition, for each field there is a corresponding flag or mask, which is used in the `GpiSetAttrs()` API: one of its parameters specifies the attributes to be changed (based on the values passed in the `LINEBUNDLE`), while another specifies the attributes to be reset to the defaults. In each case, the parameter consists of the various flags OR'ed together.

So, to set all lines drawn from this point on to be red and of default width:

```
LINEBUNDLE linebundle;
linebundle.lColor = CLR_RED;

GpiSetAttrs(hps,            // Handle to PS
              PRIM_LINE,     // type of primitive
              LBB_COLOR | LBB_WIDTH,  // attrs to change
              LBB_WIDTH,     // set to default
              &linebundle);
```

Setting Attributes Individually

Alternatively, single attributes can be changed using API's:

Lines

`GpiSetLineWidth()` - sets the cosmetic line width. This is the width used in 'normal' drawing operations, and remains the same, regardless of any scaling or zooming operations. It is expressed as a multiple of the normal device line width.

`GpiSetLineWidthGeom()` - sets the geometric line width, which is used in the `GpiModifyPath()` and `GpiStrokePath()` API's when converting lines and arcs into wide lines and arcs, which are then treated as an area. The geometric line width is specified in world coordinates, and so it changes as a result of scaling and zooming.

`GpiSetLineType()` - sets the line type: default, dotted, dash-dot, double-dot and so on.

`GpiSetLineEnd()` - specifies how line ends will be drawn: flat, square or rounded.

`GpiSetLineJoin()` - sets the line-join type: bevelled, rounded or mitred.

Areas

`GpiSetPatternSet()` - sets the current pattern-set to be used for area fills and shading.

`GpiSetPattern()` - sets the shading pattern to be used. The default set include various densities of dotted grey, and horizontal, vertical and diagonal striping.

`GpiSetPatternRefPoint()` - sets the pattern reference point (in world coordinates) so that if an area is transformed, the pattern will appear to move with it.

Markers

`GpiSetMarkerSet()` - sets the current marker set to a logical font number.

`GpiSetMarker()` - sets the current marker. The default set includes various crosses, squares, diamonds and stars.

`GpiSetMarkerBox()` - sets the size of markers selected from an outline font.

Text

`GpiSetCharSet()` - sets the current font, based on a logical font number. The font must previously have been loaded and identified using `GpiCreateLogFont()`.

`GpiSetCharMode()` - specifies the character drawing mode in an attempt to optimize character drawing.

`GpiSetCharBox()` - specifies the character box, in world coordinates.

`GpiSetCharAngle()` - specifies the angle of the text baseline, allowing a line of text to be drawn at any angle.

`GpiSetCharShear()` - specifies shearing for character drawing, to give a perspective or sheared effect to text.

`GpiSetCharDirection()` - specifies the direction in which characters are drawn, e.g. for labelling vertical axes, or drawing Chinese or Arabic character sets.

Images

All IMAGEBUNDLE attributes are global

NB All 'Set' functions have 'Query' equivalents

Segments and Retained Graphics

Segments are available in normal presentation spaces only, when operating in retain or draw-and- retain drawing mode. The beauty of retained graphics is that, rather than having to maintain data structures in your application to represent the window contents, PM will do it for you. To redraw the window in response to a WM_PAINT message, you only have to issue a `WinDrawChain()` call (although if you are smart, you will use correlation to avoid having to repaint the entire window contents, for performance reasons).

A graphics segment is a collection of picture elements - notice, segments in this context have nothing to do with Intel processor memory segments. Segments are identified by number - either assigned as symbolic constants by the programmer, or allocated from a pool in more dynamic programs such as drawing or charting applications.

Segments are generally chained together, one after another, to form the picture chain.

The real power of segments is that segments may call segments hierarchically, and the call may specify a segment transform matrix which performs a different transform on each call. So, for example, a CAD/CAM program might have a segment which represents the drawing of a bracket, and in a larger image make several calls to this segment with appropriate transforms to translate and rotate the bracket into the correct position.

Segments have many attributes: Detectability, Visibility, Chained, Highlighted, Dynamic, some of which can be automatically propagated to their children.

The elements contained in segments are calls to graphics routines - they are stored in a segment automatically as you make GPI API calls in either retained or draw-and-retain drawing modes. Elements are the editable unit, and may be referenced by name or number in insertion or replacement mode. Drawing most objects will require multiple API calls, and the element brackets allow this to be treated as a single item.

To draw an element use element brackets:

```
GpiElement(hps, lType, pszDesc);
Gpixxx( . . .);
Gpiyyy( . . .);
GpiEndElement(hps);
```

or

```
GpiElement();
```

Elements, in turn, contain one or more graphics orders. An order is a byte sequence which represents a drawing primitive. One-byte orders are a single byte which is an opcode, two-byte orders comprise an opcode and value, while long orders comprise an opcode, length byte and variable- length data. Very long orders have a sixteen-bit length field and can carry up to 64KB of data. Read the header file PMORD.H for more details about orders and the macros which are used to process them. The good news is that you generally don't need to understand orders unless you are doing low-level graphics editing (*very* low-level, using `GpiGetData()` and `GpiPutData()`)

Presentation Page Units

Coordinates can be specified in any of several units in the various coordinate spaces

Pels

PU_PELS - Screen or window coordinates. These are device-dependent, and the aspect ratio may vary

Metrics

PU_LOMETRIC	0.1 mm
PU_HIMETRIC	0.01 mm
PU_LOENGLISH	0.01 in
PU_HIENGLISH	0.0001 in
PU_TWIPS	1/1440 in

Guaranteed sizes for printers and plotters, but not displays

Arbitrary

PU_ARBITRARY	No measurement scheme, preserves aspect ratio

Coordinate Spaces

A graphical application will utilise at least three, and possibly four, coordinate spaces in which the final image is assembled. First, there is the world coordinate space. In this space, segments can be constructed to represent components of the image, using whatever coordinates are natural for the application. This is optionally transformed, by the model transform, to the model coordinate space, which is an intermediate space in which segments in different world coordinate spaces can be assembled. This is then transformed by the viewing transform, to the presentation space, and this is finally passed through the default viewing transform to the device space.

Transforms

Time to go back to school! It turns out that all the various types of transform required for a 2D graphics system - scaling, translation, reflection, shearing, rotation, etc - can all be implemented by multiplying 3 x 3 matrices. We go into the details of this in course OS291, but basically

Fonts

OS/2 provides support for two basic families of fonts. From the earliest, OS/2 has supported bitmap or image fonts, in which each character is stored as an array of pixels at a predefined resolution. The problem with such fonts is that scaling is crude: at large type sizes, the result is characters made up of large blocks, with obvious jaggies along diagonal edges. In addition, rotation and other effects are difficult at best.

OS/2 1.1 and 1.2 supported vector fonts, in which the font is defined by a series of points with straight edges between them. With lots of points, curves can be simulated, and the results aren't too bad, even when scaled up. However, OS/2 1.3 seamlessly added support for Adobe's Postscript Type 1 font technology through an integrated rasterising engine. The advantage of these fonts is that curved edges are describedin terms of sets of simultaneous equations which define a Bezier spline curve, allowing for much better definition of curves at large sizes. OS/2 Warp 4.0 (Merlin) has dropped in a rasterizing engine for TrueType fonts, which are similar in principle. Fonts which are generated through the rasterizing engine are known as outline fonts.

Remember that GPI units are often expressed as FIXED values, and this applies to font measures such as height, which is measured in printer's points (units of 1/72nd inch).

The characteristics of a particular font are represented by a FATTRS structure:

```
typedef struct _FATTRS {
  USHORT   usRecordLength;        /*  Length of record. */
  USHORT   fsSelection;           /*  Selection indicators. */
  LONG     lMatch;                /*  Matched-font identity. */
  CHAR     szFacename[FACESIZE];  /*  Typeface name. */
  USHORT   idRegistry;            /*  Registry identifier. */
  USHORT   usCodePage;            /*  Code page. */
  LONG     lMaxBaselineExt;       /*  Maximum baseline extension. */
  LONG     lAveCharWidth;         /*  Average character width. */
  USHORT   fsType;                /*  Type indicators. */
  USHORT   fsFontUse;             /*  Font-use indicators. */
} FATTRS;

 typedef FATTRS *PFATTRS;
```

The selection indicators are primarily used with image fonts in order to simulate features such as italic, bold, strikethrough, underscore and outlining, since those features are often only supported on outline fonts.

The match value identifies a particular font, and is assigned to the font when it is loaded. A negative match value indicates a device font, that is, a font which is specific to a particular device, while positive match values indicate generic fonts which will work on any device. The standard fonts, for example, are generic, as are any Adobe Type 1 fonts which are installed. The match value can be supplied to the GpiCreateLogFont() API, although more flexibility can be obtained by specifying the font name and characteristics.

The face name field is pretty self-evident, while the registry number is a unique number managed by IBM. The code page field can be defaulted in most English-speaking countries by setting it to zero.

The maximum baseline extent of the character is the maximum vertical height occupied by characters in the font. It is specified in world coordinates, and must be specified when requesting an image font with `GpiCreateLogFont()`. The average character width is the average width of characters in the font, again specified in world coordinates.

The `fsType` field is used to specify type indicators, allowing selection of fonts which contain kerning information for use on Postscript devices, as well as selection of double-byte character set code pages.

Finally, `fsFontUse` allows the programmer to specify font-use indicators which allow the GPI to optimise the font rasterization. `FATTR_FONTUSE_OUTLINE` will force use of outline fonts only, while `FATTR_FONTUSE_TRANSFORMABLE` allows the font to be used with GPI transforms, and hence scaling, rotation and other effects. This requires the current character mode to be set to `CM_MODE3`. Finally `FATTR_FONTUSE_NOMIX` means that the font will never be used to paint text across other graphics, only plain color, and allows the GPI to optimize text painting.

Printing Under OS/2 Presentation Manager

Presentation Manager operates quite differently from graphics drivers for operating systems such as DOS, which treat the screen and printer as very different devices. PM is different from earlier graphical environments in two important respects:

It handles both raster and vector devices, and

It allows for device sharing

Raster devices operate by scanning across and down the output device. Examples include most PC graphics and text displays and dot-matrix impact and many laser printers. Vector devices, by contrast, can create an image by drawing in any direction, including back 'up' the page, and would include a very few storage CRT displays, plotters and Postscript printers (conversion from vector to raster images is performed inside the printer in the last case).

PM allows for device sharing by means of overlapping windows, in the case of the screen and through a sophisticated spooler subsystem for printers and plotters.

In the case of both screen and plotter output, text and graphics output is achieved in exactly the same way: through the creation of graphics in a presentation space and association of that presentation space with a device context.

A *device context* is the means of writing output to a device. It includes both the device driver and the device itself (if any). There are several types of device contexts:

Screen device context (for example, used for menus, dialog windows, etc)

Memory device context (used for application-specific graphics representations)

Metafile device context (a portable graphic)

Other Device device context (used for printers, plotters etc.)

A subset of the 'Other Device' device context is the information device context, which is used to obtain information about the device, such as supported fonts, sheet feeders and the like.

A *presentation space* is a virtual and abstracted device context. It is essentially a 'perfect' graphics environment, in which the programmer can use function calls to create object-oriented graphics of extreme precision and resolution. The associated physical device will have to get by as best it can.

Graphic and text output is created by associating the device context with a presentation space. Once this is done, the use of Gpi drawing functions in the presentation space will cause output on the associated device context. In fact, four different modes of operation are possible:

Storing of graphics in the PS with output on the DC

Storing of graphics in the PS with no output

No graphics retained in the PS, but output on the DC

No graphics retained, no output on the DC

The latter is rather like winking at pretty girls in a darkened room - you may know what you're doing, but no-one else does and there's no end result!

With retained graphics in operation, the various Gpi function calls place graphics elements or orders - which are a kind of threaded code making function calls on the graphics primitives - into graphics segments. Segments may call other segments hierarchically, allowing hierarchical picture construction and editing, in which elements are the editable unit and may be inserted or deleted by name or number.

The GPI (Graphical Programming Interface) contains hundreds of calls for graphics and text handling. In this article, I propose to deal with just the simple problem of printing, particularly as it is not well dealt with elsewhere.

Printing under OS/2 is usually done through a print spooler which will support multiple print queues and printers. The spooler subsystem actually consist of the spooler program (PMSPOOL.EXE) which reads print jobs off its queue and hands them to the queue processor. At this stage, a print job will be in one of two formats: PM_Q_RAW and PM_Q_STD. If the print job is of type PM_Q_RAW, then the queue processor assumes that it is raw printer-dependent data (e.g. contains mixed text and escape sequences, like HP PCL 5, or straight text like Postscript) and hands it directly to the printer. However, if the print job is of type PM_Q_STD, then the print job is actually a Presentation Manager metafile, and the queue processor then invokes the printer driver to convert the metafile into something the printer can understand.

Wherever possible, OS/2 programmers writing print formatting and similar utilities should use Gpi function calls to produce output of data type PM_Q_STD, as this will work on any printer which has an OS/2 PM printer driver, and for any supported paper size (both US and metric). Programs which output printer-specific data (PM_Q_RAW) should be avoided as they are non-portable.

The first task any program which will print should perform is to find out which of the attached printers or queues the user has selected as default. This information is stored in the OS2.INI file and can be retrieved using the `WinQueryProfileString()` (1.x) or `PrfQueryProfileString()` (2.x) function call.

```
cb = WinQueryProfileString(hab,
     "PM_SPOOLER",
     "PRINTER",
     "",
     szPrinter,
     32);
szPrinter[cb-2] = '\0';
```

This will fill in the variable szPrinter with the name of the default printer, along with some semi-colons, and returns the length of the returned string. Trim the semicolons out as shown.

Having obtained the name of the default printer, the next step is to get its details, particularly the driver name and the logical port name, in order to be able to print to it.

```
cb = WinQueryProfileString(hab,
     "PM_SPOOLER_PRINTER",
     szPrinter,
     "",
     szPrnDetails,
     256);
```

The printer details is a long string which contains four fields separated by semicolons. For my default printer, it looks like this:

```
LPT1;PSCRIPT.Apple LaserWriter Plus;LWPLUS;;
```

The fields are: physical port name; printer driver and possibly other information separated by a period; logical port name; and lastly any network information (the above example is from a network, yet no network information was returned). There may be multiple values in a field; if so, they will be separated by commas and the first one will be the default.

In order to extract the printer driver name and logical port, it is necessary to parse the printer details string and extract the default values for the printer driver name (stripping out the period and any other information) and the logical port name. This is done by the functions `ExtractDriverName()` and `ExtractLogAddress()` in this example. Notice that these simple versions do not check for multiple values separated by commas.

Once this information has been extracted, the program can now open a device context for the printer. To do this, a `DEVOPENSTRUC (dop)` structure and handle to a device context (`hdcPrinter`) are required. The structure is filled in with pointers to the strings just extracted, as well as a pointer to a string which is either `"PM_Q_STD"` or `"PM_Q_RAW"`. Of course, this is a PM program, so we use `"PM_Q_STD"`. We can now go ahead and create the device context.

```
ExtractLogAddress(szLogAddress,szPrnDetails);
ExtractDriverName(szDriverName,szPrnDetails);
dop.pszDriverName = szDriverName;
dop.pszLogAddress = szLogAddress;
dop.pdriv = NULL;
dop.pszDataType = "PM_Q_STD";
hdcPrinter = DevOpenDC(hab,
              OD_QUEUED,
              "*",
              4L,
              (PDEVOPENDATA) &dop,
              (HDC) NULL);
```

The second parameter specifies the type of device context; for printing, this can be OD_QUEUED, OD_DIRECT (non-spooled - can hold up the app while printing) and OD_INFO. This last type would be used to query the printer driver without actually printing anything - for example, to query the page size when creating a new document.

We can now start the print job by issuing a DevEscape function call which will pass a "start document" message to the device context. This is used to reset the printer driver and pass data such as the document name to the spooler:

```
              lrc = DevEscape(hdcPrinter,
                       DEVESC_STARTDOC,
                       9,            /* Length of docname string */
                       "Test Doc",
                       NULL,
                       NULL);
```

Having successfully opened the device context (indicated by a non-null hdc returned), we can now associate a presentation space. There are several types of presentation space: micro, cached-micro and normal - however, a cached-micro presentation space cannot be used for printing, while a GPIT_NORMAL PS has the benefit that it can be used for screen drawing and then re-associated with the printer device context in order to produce a printed copy of the screen image. However, there are some complexities with this approach, in particular avoiding the processing of WM_PAINT messages so that screen activity does not interfere with printer output.

In this case, we'll create a new presentation space for printer output:

```
              sizl.cx = 0L;
              sizl.cy = 0L;

              hpsPrinter = GpiCreatePS(hab,
                          hdcPrinter,
                          &sizl,
                          PU_LOMETRIC | GPIF_DEFAULT | GPIT_NORMAL |
GPIA_ASSOC);
```

Notice that one of the parameters passed to the GpiCreatePS function call is a pointer to a SIZEL structure which we have previously initialized to zeros. By setting the sizl elements to zero, we make GpiCreatePS return a PS which is conveniently the same size as the printer page. What do the units in the sizl structure represent? In this case, by passing PU_LOMETRIC, we selected units of 0.1 mm, but other values are possible:

PU_ARBITRARY	Sets units to pels initially (change with `GpiSetPageViewport()` later)
PU_HIENGLISH	Sets units to 0.001 inch
PU_HIMETRIC	Sets units to 0.01 mm
PU_LOENGLISH	Sets units to 0.01 inch
PU_PELS	Sets units to pels
PU_TWIPS	Sets units to 1/1440 inch (one twentieth of a point)

The other options specify that the PS type is normal (as opposed to micro) and that it is to be associated with the device context specified.

From this point on, what the program does is up to the programmer. Any of the standard drawing functions that are discussed in texts such as Petzold will work.

When printing a multi-page document, the programmer must issue a DevEscape(hDC, DEVESC_NEWFRAME,,,,) function call to cause "form feeds" on the device driver.

Printing large (multipage) graphics images is achieved by sliding the origin of the world space coordinate system used in the application to match the coordinate space of the printer.

The following sample program does not perform any useful task, but can quite easily be modified into a file print formatter or other print utility. It also illustrates various useful techniques for debugging PM programs, including the use of the __FILE__ and __LINE__ macros to display the module name and line number where an error was encountered.

```
#define INCL_SHLERRORS
#define INCL_GPIERRORS
#define INCL_GPICONTROL
#define INCL_WIN
#define INCL_DEV

#include <os2.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "prntest.h"
#include "\prog21\lab15\pmassert.h"

MRESULT EXPENTRY MainWndProc(HWND hwnd, ULONG ulMessage, MPARAM mp1,
MPARAM mp2);
BOOL ExtractLogAddress(char * LogAddress, char * DetailStr);
BOOL ExtractDriverName(char * DriverName, char * DetailStr);

#define PMERRCHECK ShowLastError(__FILE__, __LINE__)
void ShowLastError(char * file, int lineno);
void dbuginfo(PSZ info);
int prfile(HWND hwnd, char *filename);

HAB hab;
HMQ hmq;
HWND hwndFrame;
HWND hwndClient;

CHAR szClassName[] = "Main";

CHAR szMessage[] = "Test Message";

void main()
{
    QMSG qmsg;

    ULONG ctldata;

    hab = WinInitialize(0);
    hmq = WinCreateMsgQueue(hab, 0);

    if(!WinRegisterClass(hab,
        (PCH)szClassName,
        (PFNWP) MainWndProc,
        CS_SYNCPAINT | CS_SIZEREDRAW,
        sizeof(PGLOBALS))) {
    WinAlarm(HWND_DESKTOP, WA_ERROR);
    DosExit(EXIT_PROCESS,1);
    }

    ctldata = FCF_STANDARD & ~FCF_ACCELTABLE & ~FCF_ICON;

    hwndFrame = WinCreateStdWindow(HWND_DESKTOP,
                    WS_VISIBLE,
                    &ctldata,
                    (PCH)szClassName,
                    "Test Print App",
                    WS_VISIBLE,
                    (HMODULE)0,
                    ID_RESOURCE,
                    (HWND *)&hwndClient);
    if(hwndFrame == 0) {
    WinAlarm(HWND_DESKTOP, WA_ERROR);
    WinAlarm(HWND_DESKTOP, WA_ERROR);
    DosExit(EXIT_PROCESS, 1);
    }

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

    WinDestroyWindow(hwndFrame);
    WinDestroyMsgQueue(hmq);
    WinTerminate(hab);
}


MRESULT EXPENTRY MainWndProc(HWND hwnd, ULONG ulMessage, MPARAM mp1,
MPARAM mp2)
{

    HPS hps;
    RECTL WinSize;
    char szPrinter[32];
    char szPrnDetails[256];
    char szDriverName[32];
    char szLogAddress[32];
    USHORT cb;
    DEVOPENSTRUC dop;
    HDC hdcPrinter;
    HPS hpsPrinter;
    LONG lrc;
    SIZEL sizl;
    POINTL pointl;
    PGLOBALS p;
    HWND hwndDlg;

    p = WinQueryWindowPtr(hwnd, 0);

    switch(ulMessage) {

    case WM_CREATE:
        p = (PGLOBALS)malloc(sizeof(GLOBALS));
        WinSetWindowPtr(hwnd, 0, p);
        return (MRESULT)FALSE;
        break;

    case WM_DESTROY:
        free(p);

    case WM_COMMAND:
        switch(SHORT1FROMMP(mp1)) {
        case IDM_OPEN:
            memset(&p->filedlg, 0, sizeof(FILEDLG));
            p->filedlg.cbSize = sizeof(FILEDLG);
            p->filedlg.fl = FDS_OPEN_DIALOG | FDS_CENTER;
            WinFileDlg(HWND_DESKTOP, hwnd, &p->filedlg);
            return (MRESULT) TRUE;
            break;

        case IDM_EXIT:
            WinPostMsg(hwnd, WM_QUIT, mp1, mp2);
            break;
        case IDM_QUERY:
            cb = PrfQueryProfileString(HINI_PROFILE,
                "PM_SPOOLER",
                "PRINTER",
                "",
                szPrinter,
                32);
            szPrinter[cb-2] = '\0';
            WinMessageBox(HWND_DESKTOP,
                    hwnd,
                    szPrinter,
                    "Printer Name",
                    ID_DBGI,
                    MB_OK | MB_ICONHAND);
            break;
        case IDM_QUERYD:
            cb = PrfQueryProfileString(HINI_PROFILE,
                "PM_SPOOLER",
                "PRINTER",
                "",
                szPrinter,
                32);
            szPrinter[cb-2] = '\0';
            cb = PrfQueryProfileString(HINI_PROFILE,
                "PM_SPOOLER_PRINTER",
                szPrinter,
                "",
                szPrnDetails,
                256);
            WinMessageBox(HWND_DESKTOP,
                    hwnd,
                    szPrnDetails,
                    "Printer Details",
                    ID_DBGI,
                    MB_OK | MB_ICONHAND);
            break;
        case IDM_TEST:
            cb = PrfQueryProfileString(HINI_PROFILE,
                "PM_SPOOLER",
                "PRINTER",
                "",
                szPrinter,
                32);
            szPrinter[cb-2] = '\0';
            dbuginfo(szPrinter);
            cb = PrfQueryProfileString(HINI_PROFILE,
                "PM_SPOOLER_PRINTER",
                szPrinter,
                "",
                szPrnDetails,
                256);
            if (cb == 0) PMERRCHECK;
            ExtractLogAddress(szLogAddress,szPrnDetails);
            ExtractDriverName(szDriverName,szPrnDetails);
            dop.pszDriverName = szDriverName;
            dop.pszLogAddress = szLogAddress;
            dop.pdriv = NULL;
            dop.pszDataType = "PM_Q_STD";
            hdcPrinter = DevOpenDC(hab,
                        OD_QUEUED,
                        "*",
                        4L,
                        (PDEVOPENDATA) &dop,
                        (HDC) NULL);
            if(hdcPrinter == 0L) PMERRCHECK;
            lrc = DevEscape(hdcPrinter,
                    DEVESC_STARTDOC,
                    9,
                    "Test Doc",
                    NULL,
                    NULL);
            if(lrc == 0L) PMERRCHECK;
            /* DevQueryCaps(hdcPrinter, CAPS_WIDTH, 2L, (PLONG)
&sizl); */
            sizl.cx = 0L;
            sizl.cy = 0L;

            hpsPrinter = GpiCreatePS(hab,
                    hdcPrinter,
                    &sizl,
                    PU_PELS | GPIF_DEFAULT | GPIT_NORMAL |
GPIA_ASSOC);
            if(hpsPrinter == 0L) PMERRCHECK;
            GpiQueryPS(hpsPrinter, &sizl);
            pointl.x = sizl.cx / 2;
            pointl.y = sizl.cy / 2;

            GpiCharStringAt(hpsPrinter,&pointl, sizeof(szMessage),
szMessage);

            lrc = DevEscape(hdcPrinter,
                    DEVESC_ENDDOC,
                    0,
                    NULL,
                    NULL,
                    NULL);

            GpiDestroyPS(hpsPrinter);
            DevCloseDC(hdcPrinter);

            break;
        }
        break;


    case WM_CLOSE:
        WinPostMsg(hwnd, WM_QUIT, 0L, 0L);
        break;

    case WM_PAINT:
        hps = WinBeginPaint(hwnd, 0, &WinSize);
        WinFillRect(hps, &WinSize, CLR_PINK);
        WinEndPaint(hps);
        break;

    case WM_ERASEBACKGROUND:
        return (MRESULT)TRUE;
        break;

    default:
        return WinDefWindowProc(hwnd, usMessage, mp1, mp2);
        break;

    }
    return 0L;
}

BOOL ExtractLogAddress(char * LogAddress, char * DetailStr)
{
    char *p;

    p = DetailStr;
    while(*p++ != ';');    /* Gets to first ';' and one char beyond
*/
    while(*p++ != ';'); /* Gets to second ';' and one char beyond */
    while(*p != ';') *LogAddress++ = *p++;
    *LogAddress = '\0';
    return TRUE;
}

BOOL ExtractDriverName(char * DriverName, char * DetailStr)
{
    char *p;

    p = DetailStr;
    while(*p++ != ';'); /* Gets to first ';' and one char beyond */
    while(*p != '.' && *p != ';' && *p != ',')
    *DriverName++ = *p++;
    *DriverName = '\0';
    return TRUE;
}

void ShowLastError(char * file, int lineno)
{
    char buffer[32], title[48], temp[10];
    ERRORID errcode;
    char *errormessage;

    errcode = WinGetLastError(hab) && 0x0000ffff;
    switch(errcode){
    case PMERR_OK:
        return;
    case PMERR_DOSOPEN_FAILURE:
        errormessage = "PM Err: Dos Open Failure";
        break;
    case PMERR_INV_DC_TYPE:
        errormessage = "PM Err: Invalid Device Context Type";
        break;
    case PMERR_INV_LENGTH_OR_COUNT:
        errormessage = "PM Err: Invalid length or count";
        break;
    case PMERR_INV_DC_DATA:
        errormessage = "PM Err: Invalid Device Context Data";
        break;
    case PMERR_INV_HDC:
        errormessage = "PM Err: Invalid Handle to Device Context";
        break;
    case PMERR_INV_HPS:
        errormessage = "PM Err: Invalid hPS";
        break;
    case PMERR_PS_BUSY:
        errormessage = "PM Err: PS Busy";
        break;
    case PMERR_INV_CHAR_SET_ATTR:
        errormessage = "PM Err: Invalid char set attribute";
        break;
    case PMERR_NOT_IN_IDX:
        errormessage = "PM Err: Not in IDX";
        break;
    default:
        sprintf(buffer,"Unknown error code: %08x",errcode);
        errormessage = buffer;
    }
    strcpy(title,"Module: ");
    strcat(title, file);
    strcat(title, ". Line ");
    _itoa(lineno, temp, 10);
    strcat(title, temp);

    WinMessageBox(HWND_DESKTOP,
    HWND_DESKTOP,
    errormessage,
    title,
    ID_DBGI,
    MB_OK | MB_ICONEXCLAMATION);
}

void dbuginfo(PSZ info)
{
    WinMessageBox(HWND_DESKTOP,
    HWND_DESKTOP,
    info,
    "Debug Info",
    ID_DBGI,
    MB_OK | MB_ICONEXCLAMATION);
}
```

Introduction to System Object Model

What is SOM?

The adoption of object-oriented programming techniques and languages has long held out the hope of improved productivity through increased code reuse, coupled with greater initial integrity and improved maintainability of code. However, these benefits have not been realised in the majority of cases, and for many organisations there are obstacles to the introduction of OO technology.

First is the existence of legacy code written in conventional languages such as COBOL, PL/I and C. Rather than being a problem, this code represents an asset: the code has already been paid for, it is working and may adequately serve the existing needs of the business. No-one wants to scrap this code and replace it with new code, just to get the benefits of object technology, but how can it be adapted or grafted into OO systems?

Another asset is the existing programming staff. Experience shows that attempting to retrain programmers familiar with existing procedural languages to use OO languages is often unsuccessful. In fact IBM Consulting studies show that there is approximately a 10% chance of successfully retraining a COBOL programmer in C++, although the transition to Smalltalk is more successful.

Another problem is the selection of tools. There is a wide selection of OO languages - C++, Objective C, Object Pascal, Smalltalk, Eiffel, CLOS all come immediately to mind - so that the first decision is 'what language do we use?'. This leads to another problem.

The major benefit of OO technology is the ability to buy in subassemblies or 'parts', in the form of class libraries or frameworks. After all, nobody builds a computer from scratch; the assemblers purchase cases, power supplies, disk controllers and system boards from OEMs who in turn source their processors and memory chips from semiconductor companies, who in turn purchase silicon wafers from specialist suppliers. At each stage of the process, the manufacturer is buying in subassemblies which provide a high level of functionality.

Yet in the software business, we have traditionally not worked this way. A corporation requires a new personnel system, and so a senior analyst/designer sits down with a legal pad and pencil to design the system and then programmers start coding - the whole thing being built from scratch.

OO offers the opportunity for specialists in application domains to construct class libraries or frameworks which implement business classes and behaviour. These frameworks can then be sold to customers who will write client programs to suit their business needs, and in some cases subclass the supplied classes to provide customised behaviour.

But - and here's the rub - there are no (well, few) standards for interoperability between classes in the OO world. The procedural languages have had this sorted out for years - I can write a program in C which calls library functions written in Pascal, for example - but OO is just too new. If I write a banking framework in C++, my client banks will have to use C++ for their programs too. A bank which has committed to Smalltalk will have to look elsewhere for a framework (or pay me huge chunks of cash to rewrite my code in Smalltalk!).

Even within a single language, trouble still exists. There is no standard for parameter passing and virtual function tables in C++, for example, so that client programs and classes have to be compiled with the same compiler. Even different versions of a single compiler can produce incompatibilities unless care is taken.

SOM (System Object Model) is the answer to these problems. There is an emerging standard for interoperability between classes, or more properly between objects. The Object Management Group, an international body seeking to standardise OO technology, has produced a standard called CORBA (Common Object Request Broker Architecture) and SOM is the first implementation of this technology.

SOM is 'a new object-oriented programming technology for building, packaging, and manipulating binary class libraries'[3]. How does this solve the problems described above, and what are the benefits?

Benefits of SOM

SOM and CORBA split apart the interface of a class from its implementation. The interface defines what members of the class are visible to the outside world - public instance data and methods, for example - along with their calling conventions, number and types of parameters and other information required for a client to use an object of that class.

The interface specification is written in a special language called IDL (Interface Definition Language), which in the case of SOM is CORBA-compliant with extensions. The interface is then compiled by the SOM Compiler, which should more properly be titled the SOM Preprocessor, as it produces language bindings in the form of header or include files for the target language.

The programmer then writes the code which implements the object behaviour using his choice of language - C, C++, PL/I, whatever. This is the solution to the first problem/opportunity mentioned above: existing code, written in procedural languages can be turned into classes either by 'wrapping' the existing code in a class, or by pulling apart the existing code and using it to implement the methods of the class.

In essence, SOM can be used to 'front-end' a traditional procedural programming language and give it object-oriented features. Programmers can continue to use languages with which they are familiar and can gradually adopt object-oriented techniques.

Because SOM is also a standard for binary interoperability of objects, it solves the other problems mentioned above. SOM classes can be implemented in one language and used by another. In other words, it is possible to write a SOM class in, say, C, and have it subclassed in C++, and the resultant subclass used by a client program written in Smalltalk. Thus developers can write class frameworks and sell them without having to worry about their choice of language.

This is also useful to corporate development teams. Technical and systems programmers can use SOM to implement low-level code such as client-server transport layers, while business-oriented applications programmers can use the resultant classes from higher-level languages like Smalltalk and visual programming environments such as IBM's Visual Age and Digitalk's PARTS (Parts Assembly and Reuse Toolkit).

However, for those who have a library of C++ code, their early adoption of OO techniques is repaid: the IBM Visual Age C++ compiler can generate SOM binaries directly from C++ classes and can even reverse-engineer IDL files from class source (.HH) files.

The language-neutrality of SOM comes about from four factors:

---

3 SOMObjects Developer Toolkit Users Guide, p 1-3

- SOM object methods are invoked by standard procedure calls, so that any language which can make external calls will work with SOM.

- The SOM API consists of bindings, which are header files or include files for a particular language, emitted by the SOM compiler. An emitter framework allows new languages to be added fairly easily

- Different languages use different techniques for method resolution, that is, working out which method to invoke for a particular class. SOM supports three different mechanisms for method resolution, ranging from a static technique which is similar to C++'s virtual function tables, to completely arbitrary techniques which can support interpreted languages such as Smalltalk and Object REXX.

- SOM complies with CORBA standards for data types, bindings and the CORBA-defined run-time 'Interface Repository'.

SOM provides significant benefits during development and also later in the system lifecycle as maintenance is performed. For example, thanks to its binary compatibility, it is possible for the programmer to make quite significant changes to class implementation without requiring recompilation of client programs. Generally, if the changes made to a SOM class do not require source-code changes in the client programs, then those programs will not need to be recompiled. Changes which can be made to classes, yet may not require recompilation include:

- Adding new methods

- Changing the size of an object by adding or deleting instance variables

- Inserting new base classes above a class in the inheritance hierarchy

- Relocating methods upwards in the class hierarchy to a base class.

In fact, SOM makes it easy to encapsulate and deliver classes in the form of Dynamic Link Libraries. With this technique, client programs need not even be relinked, and applications can be designed to be extensible.

For an extreme example of this, consider the OS/2 Workplace Shell user interface. This allows new classes to be registered and objects of those classes created by dragging from templates, so that new functionality can be seamlessly added to the system. It is SOM that makes this possible.

A final benefit is that SOM is also platform-neutral. IBM has implemented it on OS/2, AIX, Win16, Win32, Macintosh, various UNIX implementations, AS/400 and MVS.

SOM Class Frameworks

Compilers for languages like C++ are generally accompanied by class libraries which implement such generic functionality as I/O, complex number or vector classes and collection classes. The SOMObjects Developer Toolkit also provides a number of class frameworks.

DSOM

One drawback to conventional SOM is that, like C++, objects and their clients must exist in the same address space, as object references are passed around as pointers. For most host operating systems, this means that the objects and their clients must be in the same process, and this exposes the program system to bugs in classes. The larger the system, the more objects and classes running as part of the process, and the greater the chance that a bug in one will bring the whole process down. This could be clearly seen in the first release of the Workplace Shell in OS/2 2.0, where poorly-tested objects would regularly cause the interface to crash, although it would always restart itself and continue.

DSOM (Distributed SOM) solves this problem by allowing objects to exist in different address spaces. Objects are instantiated and run on a DSOM server process, and a client program is able to manipulate them by invoking methods on a stub called a 'proxy object' which appears in the client's address space. If either client or server crash, the other should be able to recover and continue, and a bug in one class should not affect other classes running on other servers.

Clearly, the genuine object and the proxy communicate using some form of client-server protocol. If this protocol could be transferred across a network, it would allow SOM to be used as a network client-server development tool. A subsystem called the Workgroup Enabler does exactly this: it allows objects and their client programs to communicate over NETBIOS, SPX/IPX and TCP/IP networks (and the existence of SOM for AS/400 and MVS suggests that SNA LU6.2 should also be supported).

Using SOM as a client-server protocol allows for interoperation of objects on a variety of platforms. User-interface classes running on an OS/2 workstation could interact with business classes running on an MVS mainframe. Of course, with all the debate about client-server application partitioning, SOM provides the answer: instantiate the objects wherever it makes sense.

The Interface Repository Framework

The Interface Repository is a primitive database which contains the interface information from the IDL for given classes, and provides this information at run-time for highly dynamic systems.

Replication Framework

Replicant SOM allows an object to exist in several address spaces, while actually maintaining a single image. Changes made in one addres space propagates immediately to others.

Workplace Shell

In a sense the OS/2 Workplace Shell is simply a SOM class framework. The `PMSHELL.EXE` program provides a process and context within which objects are manipulated, and each class is implemented using SOM and its class DLL registered into the system.

OpenDoc

OpenDoc is an architecture for the creation of compound, intelligent documents.

Definition of an object: "An independent piece of software that provides a specialised function and is designed to be assembled into specialised applications and/or compound documents"

OpenDoc: A robust component technology for:

- Assembling specialised applications
- Creating and managing compound documents
- Automating and integrating user tasks

OpenDoc draws heavily on SOM for its underlying functionality, and this is reflected into its robustness and utility to application developers. By contrast, Microsoft's Object Linking and Embedding (OLE) architecture is overly-complex, difficult to use, fragile and limited in functionality - for example, it does not support inheritance, which is the lynchpin of OO design. This is in part due to its genesis in Microsoft's applications division, where it was viewed originally as a facility to allow users to link together documents in Microsoft Office.

OLE has been referred to as "a morgue of dead objects" - a reference to the fact that in OLE, objects are cached but only one object can be brought to life at a time. This severely limits the utility of OLE in application development, and also limits the user interface capabilities.

How Does SOM Work?

The programmer starts by designing the class interface, i.e. the external view of the object, and defining it using Interface Definition Language. The SOM compiler uses a variant of IDL called 'SOM IDL' to distinguish it from pure CORBA IDL, and the developer can use conditional compilation to make clear which parts of the interface definition are pure CORBA-compliant and which are extensions.

Basically, the IDL file will contain the following components:

- Include statements for the IDL definitions of the base class(es) and metaclass of the new class. (In SOM, unlike C++, all classes are derived).

- Type and constant declarations

- Interface declaration(s) which specify a class name, base class(es) and the new methods, constants, type definitions and exceptions the new class supports.

The interface declarations takes the following form:

```
interface <classname> : <parent-class> {
    <constant-declarations>
    <type-declarations>
    <exception-declarations>
    <attribute-declarations>
    <method-declarations>
    <implementation statement>
} ;
```

Like C++, SOM classes can introduce new constants and types - more on this later.

Attributes are instance data members for which the SOM compiler will automatically create two accessor methods of the form `_get_<attribute-name>` and `_set_<attribute-name>`. So, for example, specifying

```
     attribute string errorMsg;
```

is equivalent to defining the two methods

```
     void _set_errorMsg(in string errorMsg);
     string _get_errorMsg();
```

Attributes may be readonly, in which case there will be no _set method.

Method declarations define the interface of each method introduced by a class, and their syntax is similar to C or C++ function declarations. However, parameters must have a directional attribute indicator (`in, out, inout`) which indicates whether a parameter is to be passed from client to server (in), from server to client (out) or in both directions (inout). For example:

```
     void dump();
     float distance(in float lat1, in float long1, in float lat2, in
float long2);
     void track(in location start, in location end, out float
bearing, out float distance);
```

A SOM IDL interface may contain an `implementation` statement, which specifies details such as version numbers, inherited method overrides, resolution mechanisms and more. Because the implementation statement is not CORBA-compliant, it should be conditionally compiled, and so the syntax for the implementation statement looks like this:

```
     #ifdef __SOMIDL__
     implementation {
          <implementation-stuff>
     } ;
     #endif
```

Typical statements encountered in the implementation statement include modifiers such as

```
     filestem = stack;
     dllname = “E:\som\utils”
     majorversion = 2;
     minoversion = 1;
```

In this case, the `filestem` statement specifies the rootname of the generated bindings files, the `dllname` statement specifies the name of the library which will contain the class’s implementation and the `version` modifiers are used to specify the version number for version control and compatibility checking.

Other important statements in the implementation section include qualified modifiers such as:

- `override` - indicates that the method is introduced in a base class and is redefined by this class

- `nooverride` - indicates that this method should not be overridden by derived classes (the SOM compiler will generate an error if this is attempted)

- `releaseorder` - specifies the order in which the methods appear in SOM’s equivalent to the C++ virtual function table. The release order should contain every method introduced by a class, including the get and set methods for attributes. The entries in the release order must be maintained in the same order - new methods must be added at the end, and entries should not be deleted, even when the corresponding method is removed. The release order is the major mechanism by which SOM ensures binary compatibility as a class is maintained and extended.

The implementation statement may also include passthru statements which allow statements to be inserted into the generated bindings files. However, this facility is provided primarily for compatibility with the older SOM version 1.0 OIDL, and new classes should limit their use to #include directives.

Comments in an IDL file can conform to the C and C++ conventions: "//" starts a line comment while '/*' and '*/' delimit a block comment. However, the SOM compiler passes comments through to the bindings files, and they must therefore be strictly associated with particular syntactic elements to ensure correct placement in the generated files. The following locations are legal:

- At the beginning of the IDL specification
- After a semicolon
- Before or after the opening brace of a module, interface statement, implementation statement or structure or union definition
- After a comma that separates parameter declarations or enumeration members
- After the last parameter in a prototype (before the closing parenthesis)
- After the last enumeration name in an enumeration definition (before the closing brace)
- After the colon following a case label of a union definition
- After the closing brace of an interface statement

SOM IDL also supports throw-away line comments, which start with "//#", do not appear in generated files and may be used to 'comment out' portions of IDL statements.

Methods or attributes within an IDL specification may be designated as private by using the preprocessor definition __PRIVATE__, like this:

```
#ifdef __PRIVATE__
     attribute int num_elements;
#endif
```

The SOM Compiler will generally recognise only non-private methods and attributes when generating the bindings file for use by client programs. In addition, a special emitter called `pdl` can be used to generate a copy of a .IDL file which has private sections removed.

Listing 1 shows a simple .IDL file which illustrates some of these elements.

```
#include <somobj.idl>
#include <somcls.idl>

interface Hello;

interface M_Hello : SOMClass
{
    Hello HelloCreate(in string message);
    // This method creates an instance of the Hello class and
    // uses the value of "message" to  initialise it.

};

interface Hello : SOMObject
{
    void sayHello();

    attribute string msg;

#ifdef __SOMIDL__
    implementation {
        //# Class Modifiers
        metaclass = M_Hello;    //# Identify Hello's metaclass
    };
#endif      //# __SOMIDL__
};
```
Listing 1. A simple .IDL file.

The SOM Compiler, `SC.EXE`, reads the class interface definition from an .IDL file and 'compiles' it to produce bindings for the target language. These are:

- An implementation skeleton (classname.C or .CPP) which contains the skeleton code for the method functions
- An implementation header file (classname.IH or .XIH) which contains macros and definitions for use by the class implentor
- A header file for use by clients (classname.H or .XH)

The SOM Compiler can also optionally produce:

- A .DEF file which is used by the linker to package a class as a dynamic link library
- A .PDL file which is a cleaned-up version of the .IDL file with no private items
- Entries in the Interface Repository allowing the class to be used dynamically

The programmer now edits the implementation skeleton to add in the executable code for the various methods. The SOM compiler will have provided *stub procedures* for the various methods, with the correct returned types and parameters, together with some default code which calls debugging routines. All the programmer has to do is complete the functions.

Rerunning the SOM compiler on the same IDL file will not overwrite the programmer's added code. Newly added methods will cause new stub procedures to be emitted, and changed parameters will cause revised function parameter lists to be generated, but the programmer's own code will be left untouched.

Acccessing Internal Instance Variables

In order to access instance variables declared in the implementation section, the programmer can use either of the following forms:

```
_varName

somThis->varName
```

Note that the somThis pointer must have been previously initialized using the `<className>GetData()` procedure, but the SOM compiler will automatically generate this statement for classes which introduce instance data. Note also that instance variables can be accessed only within the implementation of the class that introduces the instance variables, and not within subclasses or client programs (use attributes if this form of access is required).

Acccessing Parent Class Methods

The implementation header file also provides some macros for making parent method calls. This is typically required when a class overrides a parent method: it adds some processing either before or after the default processing provided by the parent class, and needs some way to call the corresponding parent class method.

This is achieved using two provided macros:

```
<className>_parent_<parentClassName>_<methodName>();

<className>_parents_<methodName>();
```

Writing a Client Program

Programs that use a SOM class are referred to as client programs. Client programs can be written in C, C++, Object REXX or other languages.

In C, the client program source must `#include` the class header file `<stem>.h`, while C++ programs must `#include <stem>.xh`. These are the usage bindings files generated by the SOM compiler for client programs.

Because the size of SOM objects cannot be known at compile time, instances of SOM classes must always be dynamically allocated.

An object variable can be declared like this:

```
<interfaceName> obj;    /* C technique */
<interfaceName> *obj;   // C++ technique
```

For example:

```
Location destination;
```

To create an object of the specified class, use the `<interfaceName>New` macro, like this:

```
destination = LocationNew();
```

When the object is no longer needed, it should be freed by invoking the _somFree method on it:

```
     _somFree(destination);
```

To invoke a method on an object in C, use the macro

```
     _<methodName>(receiver, args)
```

Example: Implementing a Stack Class in SOM

A stack class can be implemented in C++ as shown:

```
#include <iostream.h>

extern const char *msg[];

class stack {
    int *top;
    int *base;
    int size;
    void report_error(int err) { cout << msg[err]; }
public:
    stack(int insize) {base = top = new int[size=insize]; }       //
Constructor
    int full() const { return top >= base+size; }
    int empty() const {return top <= base; }
    void push(int);
    int pop (void);
    unsigned short count_items(void) {return top-base; }
} ;
```

The equivalent IDL looks like this:

```
#include <somobj.idl>
```

This is necessary because, unlike in C++, SOM objects must always be derived from some existing base class. In this case, we inherit directly from the SOMObject class, and must include its IDL definition.

```
#include <somcls.idl>
```

We must include somcls.idl because we are going to implement a metaclass, and this will be derived from SOMClass, so its definition must be #included.

```
interface Stack;
```

This is a forward reference to the interface definition for the Stack class itself; this is necessary because one of the methods of the M_Stack metaclass is a factory method which returns a Stack reference.

```
interface M_Stack : SOMClass {
    Stack StackCreate(in int insize);
    // This method creates an instance of the stack class and
initialises it from insize
    #ifdef __SOMIDL__
    implementation {
        releaseorder: StackCreate;
    };
    #endif //# __SOMIDL__
} ;
interface Stack:SOMObject {
    void push(in int value);
    int pop();
#ifdef __PRIVATE__
    attribute int *base;
    attribute int *top;
    attribute int size;
#endif
    #ifdef __SOMIDL__
    implementation {
        somInit: override;
        somUninit: override;
        metaclass: M_Stack;
        releaseorder: push, pop, _get_base, _set_base, _get_top,
_set_top, _get_size, _set_size;
    };
    #endif  //# __SOMIDL__
} ;

#include <somobj.idl>
#include <somcls.idl>

interface Stack;
interface M_Stack : SOMClass {
    Stack StackCreate(in int insize);
    // This method creates an instance of the stack class and
initialises it from insize
    #ifdef __SOMIDL__
    implementation {
        releaseorder: StackCreate;
    };
    #endif //# __SOMIDL__
} ;
interface Stack:SOMObject {
    void push(in int value);
    int pop();
#ifdef __PRIVATE__
    attribute int *base;
    attribute int *top;
    attribute int size;
#endif
    #ifdef __SOMIDL__
    implementation {
        somInit: override;
        somUninit: override;
        metaclass: M_Stack;
        releaseorder: push, pop, _get_base, _set_base, _get_top,
_set_top, _get_size, _set_size;
    };
    #endif  //# __SOMIDL__
} ;
```

The Workplace Shell and System Object Model

What is the Workplace Shell?

The Workplace Shell (WPS) is a complex beast. To the end user, the WPS **is** OS/2 Version 2. To a programmer, it is an environment in which programs can be tightly integrated with the operating system to lessen the writing of code, and still present a consistent, easy to use interface. The ability of the programmer to understand the end user is enhanced by the object oriented foundation the WPS provides.

The WPS As a Shell

Users of PC's have different levels of understanding of the processes that a computer performs in order to accomplish a task. The amount of understanding a user of a machine needs is dependent on the shell that machine provides. A shell is used to hide some of the lower level details of interacting with the operating system from the user. To operate a PC using the DOS Command Line Interface, you need to know the commands to navigate through the file system, manipulate files and the names of the application software you wish to use. For a user with little knowledge of PC's, this is a daunting amount of information to remember.

Graphical User Interfaces (GUIs) have alleviated the need for some of this knowledge by providing an easier method of navigating the file system and accessing application software. But most GUIs are just a combination of applications, each written to perform a specific function. For example, File Manager, Print Manager and the Control Panel of Windows and OS/2 1.3 are separate applications that have to be learned as any other application is learned by the end user. Granted, they provided a consistent user interface to the end user, thus alleviating some application learning, but they did not try to hide the inherent complexities of the operating system from the end user. Knowledge of directories, files and programs was still essential.

OS/2 Version 2 introduced the Workplace Shell, an Object Oriented User Interface (OOUI), which allows the end user to work with a PC with little or no knowledge of the operating system it is running, or the structure of the persistent storage strategy being used. Instead the end user interacts with familiar objects such as Printers, Folders, Documents, a Shredder and a Desktop. This allows a user who is unfamiliar with PC's to be task driven, rather than process driven. The user no longer has to think about which applications to run and which files to create to achieve the end result, rather objects are manipulated which are directly related to achieving the end result.

What about experienced users of PC's? Don't tell me we spent all this time remembering DOS commands and fast ways of doing things, to revert back to a Macintosh like interface? Luckily, no. OS/2 Version 2 also allows a user to manipulate files, directories and applications almost exactly the same way as they have been doing for years. You have the flexibility of using the WPS when you like, and still have access to the operating system on a more personal level at any time through the OS/2 Command Prompt. Most experienced users will balk at the WPS until they spend some time with it and find out the power it can provide. Some will often pop back to the command line interface to achieve a specific task, although most times this is due to the organisation of the data, not the usability of the shell itself

The WPS As a Work Place

The WPS implements the desktop metaphor of the CUA 91 guidelines with a few exceptions. The desktop provided with OS/2 Version 2's WPS is one that allows most users to manipulate objects represented by their PC, or attached to it, with very little training and guidance.

Many objects are provided for the user as part of the WPS, and almost all of them can be customised to the users particular preferences. But not everything has been provided for the user, many objects have been left to third parties to provide. For example, the Startup Folder and STARTUP.CMD file both have similar purposes. It is possible to integrate them into the one object and provide settings options to allow the user to specify that a program is to start at boot time, or at WPS initialisation time.

Users of different technology will also require different objects on their desktops. If a CD-ROM drive is connected to a machine, for example, it can be used as a storage device or an audio play-back device. OS/2 implements the first of these two in the WPS, but not the second. As a desktop, the WPS is really a clean slate - it provides all you need for general work, but allows you to extend and improve your desktop to suit your work habits.

Workplace Shell Object Hierarchy

The WPS uses a hierarchy of classes to create the objects that appear on the desktop and inside various folders. To understand and program using the WPS, you must have a basic understanding of the hierarchy and where objects should fit into it.

The WPS provides the following classes of objects according to IBM documentation :-

```
SOMObject
     |
     |-WPObject
     |
     |---WPAbstract
     |      |---WPClock
     |      |---WPCountry
     |      |---WPKeyboard
     |      |---WPMouse
     |      |---WPPalette
     |      |      |---WPColourPalette
     |      |      |---WPFontPalette
     |      |      |---WPSchemes
     |      |---WPPrinter
     |      |---WPProgram
     |      |---WPShadow
     |      |---WPShredder
     |      |---WPSound
     |      |---WPSpecialNeeds
     |      |---WPSpooler
     |      |---WPSystem
     |
     |---WPFileSystem
     |      |---WPDataFile
     |      |      |---WPInstall
     |      |---WPFolder
     |      |      |---WPDesktop
     |      |      |---WPDisk
     |      |      |---WPDrives
     |      |      |---WPStartup
     |      |      |---WPTemplates
     |      |---WPProgramFile
     |
     |---WPTransient
            |---WPJob
            |---WPPort
            |---WPPrinterDriver
            |---WPQueueDriver
```

WPS Class Hierarchy from documentation

The Toolkit that IBM provides for developing WPS applications has a slightly different hierarchy :-

```
SOMObject
     |
     |-WPObject
     |
     |---WPAbstract
     |      |---WPClock
     |      |---WPCountry
     |      |---WPDisk
     |      |---WPKeyboard
     |      |---WPMouse
     |      |---WPPalette
     |      |      |---WPColourPalette
     |      |      |---WPFontPalette
     |      |      |---WPSchemes
     |      |---WPPrinter
     |      |---WPProgram
     |      |---WPShadow
     |      |---WPShredder
     |      |---WPSound
     |      |---WPSpecialNeeds
     |      |---WPSpool
     |      |---WPSystem
     |
     |---WPFileSystem
     |      |---WPDataFile
     |      |      |---WPBitmap
     |      |      |---WPIcon
     |      |      |---WPPointer
     |      |      |---WPProgramFile
     |      |             |---WPCommandFile
     |      |---WPFolder
     |      |      |---WPDesktop
     |      |      |---WPDisk
     |      |      |---WPDrives
     |      |      |---WPStartup
     |      |      |---WPTemplates
     |
     |---WPTransient
            |---JO
            |---PDR
            |---PORT
            |---QDR
```

WPS Class Hierarchy from the Toolkit

Instead of going by the paper documentation, the information from the toolkit is assumed to be more up to date and representative of the implementation of the WPS delivered with OS/2 Version 2.

All WPS Objects are derived from the WPObject class, which in turn is derived from the SOMObject class. I will cover SOMObjects and SOM in general in the section on SOM following.

There are three main classes of objects within the WPS. Those derived from WPAbstract, WPFileSystem or WPTransient. These classes are called base classes and differ in the way that they store persistent information about objects.

WPAbstract

Objects derived from this class store state information in the OS2.INI file. As such, they can be manipulated anywhere within the work place shell, but are unable to operate without a reference to the OS2.INI file, such as when copied to a diskette and used on another machine. These objects tend to be representations of abstract concepts, and hardware whose information doesn't readily allow itself to be attached to a particular file.

WPFileSystem

Objects derived from this class store information in extended attributes associated with a file physically residing on some storage media. The .CLASSINFO EA is used to hold this information, and can be queried by any OS/2 program.

Note that the EA's are stored with the file when the High Performance File System is used, but when FAT is used as a storage strategy, all EA's for a drive are stored in 'EA DATA. SF' or 'WPROOT. SF'. This is an important consideration when you are considering the performance of the WPS.

Objects derived from this class may be moved from machine to machine, as long as the EA's associated with that object are moved as well.

As you can see from the hierarchy diagram, there are currently two classes of objects derived from WPFileSystem :- Data Files and Folders.

WPTransient

Objects of this class are created 'on the fly', and destroyed after the application or object that creates them is finished processing. Examples of this are print jobs, which appear in the Printer object that is usually installed on the desktop of the WPS. These print jobs are accessible as objects, but are destroyed from the WPS Printer object once the information has been printed. If the object or application 'containing' transient objects doesn't reinstate them when a machine is powered down, they will be lost. Print jobs are recreated from entries in the spool directory associated with that printer, and the driver being used to print the information.

Implementing New WPS Objects

This subject will be covered in more detail in the section 'Creating an object - a programmers view', but it is worth mentioning that to implement new objects, you need to understand the WPS Hierarchy and how your new object fits into this scheme.

Creating a new base class for the WPS is not an exercise to be taken lightly, and given the base classes provided, most new objects will naturally belong in one particular area. For example, our Audio CD Player is probably best integrated into the WPAbstract class, as it bears no particular physical representation to the storage used in the other base classes, unless of course, the tracks on the CD are seen as files, in which case it may be better as a WPFolder with special objects inside it.

The Workplace Shell and Presentation Manager

WPS applications and PM applications are really beasts of different species, although it is possible to integrate PM applications with the WPS by providing WPS-like facilities.

The WPS as a PM Program

The WPS itself is just a PM program that has been created using SOM, and as such, follows the same drag and drop protocol as PM, uses the same types of resources, and creates the same PM objects such as windows, bitmaps and icons.

However, the WPS and any objects it loads, currently run as a single process under OS/2 Version 2. None of the usual interprocess protection that OS/2 provides is of any use, if the object is a single threaded process. This means that any object in the WPS can cause the entire shell to coming to a screaming halt. This will be changed in a future release of the WPS, although good design dictates that if features are provided for interprocess communication and protection, and they are applicable, they should be used. This means that if a WPS object is involved in say retrieving information from a database, it should create two separate processes, one to act as a client on behalf of the WPS process, and the other to act as a database server. The client process then communicates with the WPS process using any form of interprocess communication.

A WPS object is a DLL that can contain resources and code for the WPS to invoke. The resources are loaded as PM resources are loaded from any DLL, using DosLoadModule or DosGetModuleHandle. Fonts, bitmaps, icons, pointers and any other form of resource can be stored in the object's DLL and loaded at the appropriate time.

Migrating PM Programs to the Workplace Shell

Ideally, from an end user's perspective, all applications would be instantly rewritten as a collection of objects for the WPS, tightly integrated, and easy to use. Unfortunately, this costs time and money :- something not that many companies are willing to throw away without a large justification. The benefits of the WPS approach have to be weighed up carefully, and a sensible decision made.

Things you can do without totally rewriting

- Associate the application with any of the data files it uses via the Settings of the program reference object. This means that a user can automatically invoke the application by double-clicking on a data file, and not need to know the applications executable name. This means that the program will have to accept a file name as one of its arguments. Another possibility is the use of an ASSOCTABLE statement when building the application. This is covered below.

- Replace menu bars with context menus using WinPopUpMenu. This provides a more consistent interface with the WPS.

- Look for dialogues which you could replace with the new controls such as Sliders, NoteBooks and container objects. These new controls tend to make the application easier to use and seem more like a WPS object.

- Make sure the resource file for your application has an ASSOCTABLE statement that provides the default associations, rather than the user doing it. This also will provide the user with a template data file in the systems Templates folder (see example below).

- Support some form of direct manipulation if possible. This can be a large amount of work in some instances, so be careful when choosing to do this. A good example is to allow to user to drag information to the printer, this can be extremely useful to the end user.

- Implement options that allow the user to tailor the application, as a Settings view would. These options could be presented in a notebook control.

- If you will be letting the user create data files from templates, make sure your application can handle empty files.

- Have an automatic procedure to create the program reference object on the desktop, or in its own folder, alleviating the need for the user to do this. This will be covered in the section on WPS and REXX.

Using an ASSOCTABLE Statement to Provide New File Types

```
ASSOCTABLE 1 PRELOAD
BEGIN
     "123 Worksheet", "*.wk*", EAF_DEFAULTOWNER, 123.ico
END
```

The first quoted string is the File Type that appears in the associations page of the program reference objects settings.

The second lets you associate the application by a file mask, rather than using the templates. This string may be empty if you don't want to do this.

The third parameter lets the system know that we want our application to be automatically invoked when one of the aforementioned types of files is opened.

The last parameter is an icon to be used for any data files matching this description.

Considerations when moving to a WPS implementation

- Most application specific code can be reused. A WPS object still uses PM to load resources and create windows, so try not to throw out your hard work.

- Look for opportunities to put code in a separate process, and remember the one process nature of most WPS objects. This should lead you to some form of interprocess communication method which should be consistently used throughout the object's implementation.

- Take another look at the user interface. New controls may be used, and are more easily manipulated using WPS. Similarly, look for opportunities to use direct manipulation. Container windows, icons and folders can often replace listbox controls and are more user friendly

- Consider what sorts of views of the data the user may want.

- Take advantage of the ability of an object to save its state, and use that information to have the object appear as it has done previously when it is reopened.

- Beware the Minimized Window Viewer and OS/2 Version 2's new minimisation strategy. You may need to provide the user with a way of accessing minimised windows or objects, as the operating system now 'hides' minimised windows and if the window is not in the window list, where is it?

The System Object Model

SOM is a language neutral, object oriented programming interface for the implementation, manipulation and sharing of software objects. SOM is a mouthful.

SOM was provided as a way of adding and modifying objects and interfaces to those objects to an operating system in a wide variety of languages. Interfaces between objects provided by the system and those supplied by others, had to be efficient and consistent, so that even if a new version of the operating system were to be released, third party objects would still be usable.

SOM is really an object and class definition language. As such, it is not used to define the **methods** associated with classes and objects, and as such remains impartial to the language that the object is implemented in. SOM allows the original implementer of an object, and later users of that object to program in totally separate languages. The reasons for SOM's language neutrality are

- Interaction with SOM is done through standard (ANSI) procedure calls.
- SOM classes support several forms of method resolution, which correspond to a wide range of commercial object oriented programming languages.
- The SOM API can be implemented differently for different languages via SOM bindings. This is the ability of a program to produce native code from a SOM source file.

Method Resolution

Method resolution is the way in which the object oriented language works out which particular piece of code should be called when the objects method is invoked. SOM supports three ways of method resolution :-

- Offset resolution : Similar to C++ virtual functions - a part of the object's interface.
- Name resolution: Similar to Objective C and Smalltalk in that it is dynamically resolved.
- Dispatch Resolution: Used for distributed objects and the like. Only the receiving object knows whether the method is valid.

Classes Supplied With SOM

SOMObject

This class defines what an 'object' is to SOM. It holds the behaviour common to all objects, and is the class that all other classes are derived from. This class has no instance data, but several methods to define the behaviour of objects.

SOMClass

Objects respond to instance methods which manipulate the data associated with that particular object instance. SOM classes are real objects and as such are instances of some class. This sort of class is called a metaclass. An object's methods are defined in its class description, and a class's methods are defined in its metaclass description.

This is the base class for all metaclasses. It defines the behaviour of all SOM Class objects. SOMClass is both unique and confusing in that it is its own metaclass. This takes quite some thinking to get used to. An object of class SOMClass is a class object which provides constructor methods and informational class methods for that particular class.

SOMClassMgr

The SOMClassMgr object is the object that handles the registry and loading of classes from DLLs for each process that uses SOM. Only one instance of this object is created per process. SOMClassMgr is not usually used by programmers as a parent class.

How Can I Use SOM?

There are really two answers to this question :-

If you are using an OO language now

You can use SOM as a standard for importing existing classes in other languages into your own environment, or exporting your classes for use by others. SOM makes its possible to access classes (or subclass other classes) in other languages through the one interface, a special interface does not have to be written for each object in each language you wish to use. To export your own languages classes, you create corresponding SOM classes and methods, which simply invoke your native methods. Other users of your objects only ever go through SOM to interact with your class of objects.

If you are not using an OO language now

You can use SOM to turn your language into a fully object oriented programming language. SOM 'bindings', which are a set of macros and/or procedure calls are used to access and manipulate SOM objects.

What generally happens is that when an object is implemented, code for a particular language may be generated so that language can access objects. The generation of this code (SOM bindings), is tied to the SOM Compiler. Currently only the C programming language has a full set of SOM bindings, but other language vendors are considering supporting SOM and will be more inclined to do so with customer support.

The SOM Runtime Environment

What sort of overhead do you pay for using SOM?

The SOM run time environment is one DLL - SOM.DLL, which contains the three SOM Classes, and related functions for initialising SOM and customising the SOM environment.

When SOM is initialised, it creates four objects :-

- The SOMClass class object for creating other classes
- The SOMObject class object for creating SOMObjects
- The SOMClassMgr class object for creating the SOMClassMgr object to assist in loading and registering SOM classes
- The SOMClassMgr object

SOM is fully re-entrant and can be used from multiple threads concurrently. Suppliers of multi-threaded programs however should ensure that they provide synchronisation for objects shared across threads and processes using mutual exclusion semaphores or a similar method.

The Workplace Shell & SOM

The WPS uses SOM to implement its class hierarchy as described in the previous section WPS Hierarchy. As such, the WPS is the first SOM application sold, and can be used as a good example of how to use SOM to implement complex concepts. To make the most of the WPS requires careful study of the methods that are supplied with each object and careful consideration on which you would override.

WPObject

All WPS objects are derived from the WPObject class and inherent the many methods it defines. Among these are methods to :-

Set and query an object's title

Set and query an objects icon

Set and query the default view of an object

Save and restore state data for the object

Open and close an object

Add settings pages, pop up menus

Handle drag and drop protocols

This class defines the basic attributes of any WPS object and overriding these methods defines the look, feel and behaviour of an object you may create.

WPS Runtime Environment

The WPS being a SOM application creates SOMClass objects for each class that is registered to the shell, and then creates the actual object itself from the class object. This is a slight overhead, but considering the objects provided and the integration, well worth the cost.

The WPS is made up of an executable and several DLLs. PMSHELL.EXE is the executable program that loads the WPS, and PMWP.DLL is the DLL which holds the WPS Class libraries.

Applications vs Objects

The Windows and OS/2 V1.x interfaces were application oriented. The user set up groups of applications and used the GUI to launch each application. A small amount of integration was provided using the File Manager and some drag and drop facilities. This sort of interface required the user to learn several small but well designed applications and to tie these together to perform tasks.

The WPS is an object oriented interface and presents the user with objects to manipulate that are directly involved in the task the user wishes to perform. As well as supporting objects, the WPS also allows applications to be integrated into it, and provides some basic facilities for making applications seem more object oriented. Examples of this are the associations and templates that can be used with straight OS/2 PM applications.

The user of a WPS developed object needs to be able to identify with the object readily, as well as find it intuitive to work with. More than ever, the user interface is the application. Programmers have now been given the job of Object Designers and will need to spend far more time thinking about what a user of an object would want and expect.

Design Considerations

- What views of the object might the user want? What would be the default view?
- What settings of the object should there be?
- What would the user want to be able to do with the object?
- What options should be available to the user without opening the object?
- Where and how should context menus be provided?
- How does the object integrate with other common desktop objects?

Programming considerations

- Is this a new base class or a derived class?
- What would the parent class be?
- Which methods of the parent classes should be overridden and how?
- What new methods should be public?
- What instance data should be public?

Deciding on an application versus a fully fledged WPS Object is a design consideration and should take into account many different views, performance, usability and maintenance among them.

Using Objects (an end users view)

It is usually quite easy for an end user to work with the WPS after a short period of time, although some things can be done to make life a little easier.

Creating Folders

A folder with related objects in it is often something that end users will create after quite some time using the WPS. A work area is something that most will not even find until they are told about it. A work area can be used to represent a task that the user performs. All the necessary templates can be placed in the work area and associations set up for the objects that those templates become.

Commonly Used Objects

Most commonly used objects should either be shadowed onto the desktop (in the case of data files) or moved there from nested folders.

If you do not want an end user to be able to delete a data file, only provide a shadow of the file for them to use, not the original object.

Using Applications

Although the WPS provides an object oriented interface, not all the software an end user uses in the day to day hustle and bustle will be WPS aware or even OS/2 software. Care should be taken to show the user the different types of applications and how best to interact with them.

Many users are sometimes confused when they are used to pressing the right mouse button over a window and find that no menu comes up. This inconsistency is one of the hardest to explain, as to the end user, all objects are equal.

The Work Place Shell and REXX

REXX (Restructured Extended Executor) can be used to manipulate various objects in the WPS and even to register and deregister objects from the system.

Registering Objects

Using the two provided functions (SysRegisterObjectClass and SysDeregisterObjectClass), objects may be installed using a .CMD file as part of an applications installation process. This lets the implementer of an application install the object DLL's and immediately provide them for use.

```
/* */
Call RxFuncadd 'SysLoadFuncs', 'RexxUtil', 'SysLoadFuncs'
Call SysLoadFuncs
Call SysDeregisterObjectClass "WpConfig", "OldDll"
If SysRegisterObjectClass("WPConfig", "Config") Then
   Say "Config.Sys Editor Registered"
Else
   Say "Config.Sys Editor failed to register"
```
Sample Code to de/register an Object Class

Creating Objects

Using REXX, you can create folders, shadows, program reference objects, and even 'run' objects by creating them with a default open view.

```
/* FULLFOLD.CMD */
/* Builds a folder on the DeskTop and places some program objects in
it.*/

call RxFuncAdd 'sysloadfuncs', 'rexxutil', 'sysloadfuncs'
call sysloadfuncs

call SysCls
Say '';Say 'Using REXXUTILs to Add a Folder and Program Objects...'

classname='WPFolder'
title='Test Folder'
location='<WP_DESKTOP>'
setup='OBJECTID=<TEST_FOLDER>;'
Call BldObj

Say ''
Say 'Now open the folder, currently no items are there.'
Say 'Press a key to continue...'
key=SysGetKey()

Say 'Place a program object into the folder...'
Say '';
classname='WPProgram'
title='SYSLEVEL-FULLSCR'
location='<TEST_FOLDER>'
setup='PROGTYPE=FULLSCREEN;EXENAME=\OS2\SYSLEVEL.EXE;',
   'OBJECTID=<TEST_SYSL>;'
Call BldObj

classname='WPShadow'
title='SysLevel-Shadow'
location='<TEST_FOLDER>'
setup='SHADOWID=<TEST_SYSL>;'
Call BldObj

classname='WPProgram'
title='CHKDSK-PM'
location='<TEST_FOLDER>'
setup='MINIMIZED=YES;PROGTYPE=PM;',
   'EXENAME=\OS2\PMCHKDSK.EXE;OBJECTID=<TEST_PMCK>;'
Call BldObj

classname='WPProgram'
title='SYSLEVEL-VIO'
location='<TEST_FOLDER>'
setup='PROGTYPE=WINDOWABLEVIO;',
   'EXENAME=\OS2\SYSLEVEL.EXE;OBJECTID=<TEST_SYSLVIO>;'
Call BldObj

classname='WPProgram'
title='MEM-Fullscreen'
location='<TEST_FOLDER>'
setup='PROGTYPE=VDM;EXENAME=\OS2\MDOS\MEM.EXE;',
 'PARAMETERS=/?;NOAUTOCLOSE=YES;OBJECTID=<TEST_MEMFUL>;'
Call BldObj

classname='WPProgram'
title='MEM-WindowVDM'
location='<TEST_FOLDER>'
setup='PROGTYPE=WINDOWEDVDM;EXENAME=\OS2\MDOS\MEM.EXE;',
  'PARAMETERS=/?;NOAUTOCLOSE=YES;OBJECTID=<TEST_MEMWIN>;'
Call BldObj

Say '';Say 'All done, to remove objects drag to shredder...'

Exit

/* Build Object */
BldObj:
call charout ,'Building: 'title

result = SysCreateObject(classname, title, location, setup)

If result=1 Then call charout ,'...   Object created!'
Else             call charout ,'...   Not created! Return
code='result

Say '';
Return
```
Creating a folder and objects using REXX

Running WPS Objects

It is possible to create an object and get it to open immediately. This gives the effect of 'running' a WPS object from a batch file. It is useful for being able to start a Windowed VDM for instance from a batch file, for a program that cannot be run from a DOS Command prompt.

```
/* Boot a Dos image with defined sessions */
call rxfuncadd 'SysLoadFuncs', 'REXXUTIL', 'SysLoadFuncs'
call SysLoadFuncs
call SysCreateObject 'WPProgram', 'Booted DOS', '<WP_NOWHERE>',,
   'EXENAME=*;PROGTYPE=WINDOWEDVDM;STARTUPDIR=A:\;'||,
   'SET DOS_STARTUP_DRIVE=C:\WORK\DOS40.IMG'||,
   ';OPEN=DEFAULT',,
   'REPLACE'
```
'Running' WPS Objects

Note the use of `OPEN=DEFAULT`. This is the secret to getting the object to run.

The `<WP_NOWHERE>` which isused as the location for the object is a special location under the WPS, it is a place where objects are destroyed once they are closed.

The `DOS_STARTUP_DRIVE=C:\WORK\DOS40.IMG` is a file which has been created via VMDISK to be an image of a DOS session captured from floppy.

OS/2 Warp Programming Course Lab Exercises

The Lab Exercises are supplied in the form of a floppy disk. To install them, insert the floppy disk into drive A: on your machine, open a command prompt, then give the following commands (user input is underlined - substitute another target drive if C: is not appropriate)

```
[C:\]A:
[A:\]INSTALL C:
```

This will create a subdirectory tree called \OS290 on the target drive. Underneath this will be a series of subdirectories called \OS290\LAB1, \OS290\LAB2, and so on.

In these you will find the source code for the lab exercises. In most cases, I have taken a working program and excised some of the code, replacing it with comments instructing you to insert the appropriate code. If you find six blank lines, then I have probably removed five lines of code - if you can get the program to operate by inserting one line, well done! - but if you find yourself writing twenty lines of code, you're probably missing an easier way of doing the job!

To find these missing blocks of code, open the source code file and search for the string "*LAB*" as this is always inserted at the beginning of the comment.

Lab 9, however, has no LAB9.C file, as in this lab, the exercise is to design the program and write it from scratch.

Also in the lab exercise subdirectories, you will find .SOL files; these are the solutions to the labs. Try not to look at the solution until you've given the exercise your best shot! In at least one case (Lab 9) the solution is encrypted, so you can't cheat, anyway. . .

And finally, the subdirectories also contain the .MAK files which are used on my system to build the working programs. Since, on my system, the labs exist on the E: drive, there may well be incorrect drive references when installed on your system. It's usually best to rebuild the make files on your system - in fact, I want you to learn how to do this so you will know the various compile and link options required for building DLL's, multithreaded programs, optimisation and so on.

Lab 1. Introduction to the Programmer's Workbench and a Basic PM program

In the subdirectory \PROG21\LAB1, you will find the source code for a simple "Hello World!" program, HELLO.C.

1. Create a Project for this program, to generate a debuggable OS/2 Presentation Manager program.

2. Compile the program and check that it works. View the compiler output in a window.

3. Run the IPMD debugger and single-step through the program. Observe what happens when you reach the event loop. Set a breakpoint at the first statement of the WM_PAINT processing and check that a valid presentation space handle is being returned.

4. Invoke the icon editor and use it to create an icon for the program. Create a resource compiler script consisting of the lines:

ICON 100 PRELOAD iconfile.ico

and include it in the file list. Rebuild the application, run it and see if your icon appears when the program is minimised. What changes might have to be made to the source code?

Lab 2: Menus and Messages

Add a menu to the "Hello World" program to allow the user to choose the message which is displayed.

Try a cascading menu.

Lab 3: Business Forms

In the directory \PROG21\LAB3, you will find the files LAB3.C, LAB3.RC, LAB3.H, and other supporting files. This is the source code for a simplfied version of a program which is used to maintain a database of navigation aids and airports for use in flight planning.

The program makes use of a set of forms-handling functions in the subdirectory \PROG21\FORMS, files FORMS.C and PMFORMS.C. It also uses a dynamic link library called OPENDLG.DLL, and you should ensure that this file is in your LIBPATH somewhere.

1. Make up a project for this lab exercise.

2. Locate the missing sections in the file LAB3.C by searching for the string "*LAB*". Read the comments and add the required code.

3. Check the operation of the push-buttons when you append to the file. Do they work correctly?

Lab 4: Memory Management

LAB4.C provides the skeleton of a program which will ask the user his name and then echo it back.

1. Create a project for this program and set the compile and link options.

2. Locate the missing sections of code and insert code to allocate a memory object and use it to store the user's name.

3. Does the program work? If not, why?

4. Modify the program to use malloc() and, if time, to use the heap management functions (DosSubSetMem(), DosSubAllocMem(), etc.).

Lab 5: Dynamic Link Libraries

The files LAB5MAIN.C and LAB5DLL.C contain a simple program consisting of two separate modules. Read them briefly and compile and link them to check the operation of the program. Notice that LAB5.MAK compiles and links the two files into one executable, LAB5MAIN.EXE

1. Create projects for the targets LAB5MAIN.EXE and LAB5DLL.DLL. Edit the source code as required. Compile and link these files and test the program.

2. Create a composite project for the two targets. Can you compile them both simultaneously? If not, which should be compiled first?

Lab 6: Threads

The file LAB6.C contains most of the code for a simple demonstration of multi-threading. It also demonstrates some of the Vio function calls. Read the comments in the program file and insert the missing code to turn the program into a properly multi-threading program.

Lab 7: Semaphores and Shared Memory

The subdirectory \PROG13\LAB7 contains source code for two programs, called LAB7SEND.EXE and LAB7RCV.EXE.

1. Read the program comments and add code where appropriate to allow the two programs to communicate through named shared memory.

2. If time, modify the programs to use giveaway shared memory.

Lab 8: File Handling

The subdirectory \OS290\LAB8 contains source code for a directory listing utility

Lab 9: Multiple windows and instance data

This is a major lab session, which can require an entire day to complete, but brings together the concepts learned so far. It also introduces the student to PM program design.

In the subdirectory \PROG21\LAB9 you will find a program called LAB9GOAL.EXE. Run it and review its behaviour. Now, design and write a program to duplicate its functionality. To help you and to save time, we have supplied LAB9.RC and LAB9.H. You just have to write LAB9.C.

Hints: review LAB2.C, but do not attempt to simply modify it to achieve the desired effects. You will find that you will be much more productive if you start from scratch and design your program from scratch using paper and pencil, rather than hacking at an existing design.

You may notice that the LAB9 directory contains a file called LAB9.SOL and think that all your troubles are at an end. However, you will find that LAB9.SOL has been encrypted and you will receive the password late in the lab exercise, to assist those who are irretrievably stuck. Warning! Do not attempt to decrypt LAB9.SOL - this will probably double-encrypt it, and you will never be able to decrypt it again!

Things to think about:

How many windows are there (not counting controls, menu, etc.)?

Which window owns the menu?

Which window should really get the WM_COMMAND messages from the menu?

How much should one window know about another window's business?

Where should you put the code for creation of the child windows? In the main() function? Where else?

How big is the main window when the children are being sized and positioned relative to it?

Do you want the children resized every time the parent is resized?

Should HAB hab be declared static and global? What if you put the code for the child windows in a DLL to encapsulate it?

Useful Functions - in alphabetical order

These are not the only functions you can use to solve this problem, not need you use all of them!

```
WinAlarm()
WinBeginPaint()
WinCreateMsgQueue()
WinCreateStdWindow()
WinDefWindowProc()
WinDestroyMsgQueue()
WinDestroyWindow()
WinDispatchMsg()
WinDrawText()
WinEndPaint()
WinGetMsg()
WinInitialize()
WinInvalidateRect()
WinLoadString()
WinPostMsg
WinQueryActiveWindow()
WinQueryAnchorBlock()
WinQueryWindowPtr()
WinQueryWindowRect()
WinRegisterClass()
WinSendMsg()
WinSetWindowPos()
WinSetWindowPtr()
WinTerminate()
```

Useful Messages - in alphabetical order

```
UM_INITSIZE
WM_ACTIVATE
WM_CLOSE
WM_COMMAND
WM_CREATE
WM_DESTROY
WM_ERASEBACKGROUND
WM_PAINT
WM_QUIT
```

Lab 10: Dialog windows

The file LAB10.C contains a more sophisticated version of the Navaid Maintenance program examined in Lab 3. Add a dialog window which will allow the user to search for a location either by its full name or its abbreviation. Functions which will search the file are provided for you.

Lab 11: File and Font Dialogs

Examine the file \PROG21\LAB11\LAB11.C and determine its operation. Locate the *LAB* comments and insert the correct code to display the standard file and font selection dialogs.

Lab 12: GPI Functions

Run the program \PROG21\LAB12\LAB12.EXE to see what it does. Examine the source file \PROG21\LAB12\LAB12.C, analyse its operation and insert the missing code to complete the application.

Lab 13: Fonts

The subdirectory \PROG21\LAB13 contains another version of the simple editor used in earlier labs. Edit the file LAB13.C and locate the *LAB* comments. Insert the code required to complete the application.

Lab 14: SOM

The subdirectory \PROG21\LAB14 contains SOM code for the implementation of a stack class. The implementation is incomplete - you will have to write the method stubs.

Lab 15: Free Time Lab Exercises

The subdirectory \PROG21\LAB15 contains miscellaneous sample programs and additional exercises:

SUBCLASS.C Example of subclassing the listbox to add support for menu choices

OWS.C & OBJECT.C Example of an object window being used to implement a time delay without interfering with the PM synchronous message queue operation.

Common OS/2 & PM Programming Errors

1. Remember that OS/2 Dosxx API's use parameters as both inputs to the function (call by value) and outputs from the function (call by reference). Should you be passing a handle, or the address of a handle? A PVOID or a PPVOID?

2. Remember to initialise arguments to functions, even when they are being used as outputs (example: DosOpenEventSem()).

3. Remember to initialise structures, most commonly using the `memset()` function. Remember also to initialise any '.cb' structure element.

4. Remember that C uses '=' for assignment, but '==' for comparison, but often either is syntactically acceptable and so the compiler can't flag the error. It's sometimes worth taking a couple of minutes before compiling to use your editor to search for '=' and checking that that's really what you meant. Another way of avoiding the problem is to place a non-lvalue on the left of the '==', as in

```
while(TRUE == more) {
     . . .
}
```

If you should omit one of the equals signs, the compiler will now complain that an lvalue is required to the left of an equals.

5. (For OS/2 1.x code) Remember that when an application is compiled in small model, the C library functions will expect to be passed _near pointers. Data stored in segments allocated with DosAllocSeg() and DLL data segments can only be referenced with _far pointers, and these will cause errors and exceptions when passed to the functions which expect _near pointers.

Investigate the special 'far argument' library functions such as _fstrcpy(), _fstrcat(), etc.

Avoid C library functions - use the base OS functions instead, which are always _far.

If in absolute despair - as a last resort - compile in large model. Or better still, port to OS/2 2.X.

6. If `WinCreateStdWindow()` returns a zero `hwndFrame`, indicating failure to create the window, check for the following errors:

   a) Are your frame creation flags set correctly, or are you calling for creation of a non-existent resource, such as an ACCELTABLE?

   b) Have you provided the correct resource ID.

   c) Is the `WM_CREATE` processing returning (MRESULT) 0?

   d) Did you remember to resource compile the resources onto the .EXE or .DLL?

7. Remember that Presentation Manager programs do not execute from top to bottom. Instead, they are event driven. Do not assume that because you have assigned a value to a variable higher up the page, that that code has actually executed. In PM, a common mistake made by beginning programmers is to assume that hwndFrame was set by the `WinCreate(Std)Window()` API call, and to then use it in `WM_CREATE` processing. But, of

course, when `WM_CREATE` is being processed, `WinCreate(Std)Window()` has not yet returned and hwndFrame has not been assigned a value! Instead, use the WinQueryWindow() and WinWindowFromID() API's to work out window handles at run time. This also increases modularity and increases the probability of your being able to reuse window procedures.

Key Concepts

1. A window is an object. The window procedure provides the class methods for the window class, and the window words provide storage for instance data behind the window. The window is the fundamental unit of application design.

2. Do not attempt to use functional decomposition. Instead, use decomposition into windows, each of which implements a part of the user interface.

3. Windows do not have to have on-screen appearance. A window is often a useful way of implementing user interface logic by coordinating its child windows. Dialog windows are the most obvious example of this, but application clients can also function this way.

4. Use threads. The primary thread of your application can perform user interface logic, while secondary threads are used for application business logic, slow window repainting, and keeping the system responsive. Do not be afraid of threads.

Tests

Intel Processor Architectures

1. The 80286 processor is a 32-bit processor.

    a) True

    b) False

2. The 80386 processor can only support 64 KB segments.

    a) True

    b) False

3. The maximum address space of a 32-bit OS/2 application is:

    a) 640 KBytes

    b) 16 MBytes

    c) 512 Mbytes

    d) 4 GBytes

4. The 386 processor swaps pages which are:

    a) 512 Bytes

    b) 4 KBytes

    c) 64 Kbytes

    d) 16 MBytes

5. 32-bit code is faster than 16-bit code.

    a) True

    b) False

    c) It depends

    d) I'll take two

6. 32-bit OS/2 applications are written in Small model.

    a) True

    b) False

7. The Pentium's AX register is 32 bits in size.

    a) True

b) False

8. The Pentium's CS register is 16 bits in size.

a) True

b) False

9. The feature which allows OS/2 Warp to multitask DOS and Windows applications is called:

a) The DOS Compatibility Box

b) 8086 Virtual Machine Mode

c) Windows 95

d) 386 Enhanced Mode

10. OS/2 applications run in

a) Ring zero

b) Ring one

c) Ring two

d) Ring three

Introduction to Presentation Manager Programming

1. The library of graphical user interface functions in OS/2 is called

   a) The Workplace Shell

   b) Presentation Manager

   c) The Graphical Programming Interface

   d) Windows

2. Presentation Manager programs get their input by

   a) Calling gets(), scanf() and other C functions

   b) Calling the WinGetInput() function

   c) Reading messages from an input queue

   d) Using INT 33H for mouse input and INT 16H for keyboard

3. Presentation Manager create a message queue

   a) For each window of your application

   b) For each application

   c) For the entire system

   d) For each message

4. A window is

   a) A data structure inside PM

   b) A rectangular area on the screen

   c) A box on the screen with titlebar, menu, size border, system menu and scrollbars

   d) A data structure inside PM, plus an associated window procedure

   e) An object

5. A queue message is

   a) A 32-bit symbol

   b) A target window handle and message symbol

   c) A target window handle, message and two mparams

   d) A window handle, message, two mparams, timestamp and mouse position

6. The WinInitialize() API returns:

   a) A handle to an anchor block

   b) A handle to a presentation space

   c) A handle to a device context

   d) A handle to a window

7. A Presentation Manager application is a single window.

   a) True

   b) False

8. The FCF_STANDARD constant for frame creation flags includes

   a) FCF_ICON, FCF_VERTSCROLL and FCF_TASKLIST

   b) FCF_ICON, FCF_ACCELTABLE and FCF_TASKLIST

   c) FCF_VERTSCROLL, FCF_TASKLIST and FCF_ACCELTABLE

   d) FCF_SIZEBORDER, FCF_SHELLPOSITION and FCF_HORZSCROLL

9. The WinCreateStdWindow() API

   a) Creates a frame window

   b) Creates a client window

   c) Creates a desktop window

   d) a) and b)

   e) a) and c)

10. The returned type of a window procedure is

    a) EXPENTRY

    b) ULONG

    c) MRESULT

    d) int

11. The WM_CLOSE message

    a) closes the target window

    b) shuts down the system

    c) tells you the user wants to close the window

    d) tells you the user wants to quit the application

12. An MPARAM is

    a) A message parameter which can be interpreted different ways for different messages

    b) A 16-bit pointer to void

    c) A 32-bit window handle

    d) A 32-bit queue handle

13. A resource is

    a) A help file or on-line documentation

    b) A window

    c) A dynamic link library

    d) A binary icon, bitmap, menu string, acceltable or dialog

14. You can force your window to repaint by posting it a WM_PAINT message.

    a) True

    b) False

15. The WinBeginPaint() API returns:

    a) an error code

    b) a window handle

    c) a presentation space handle

    d) a device context handle

16. The event loop of a PM program consists of the following statements:

    a) while WinGetMsg() WinDispatchMsg()

    b) while WinPeekMsg() WinDispatchMsg()

    c) while WinGetMsg() WinSendMsg()

    d) if WinGetMsg() WinSendMsg()

17. If you choose not to process a particular message, you should

    a) return TRUE

    b) return FALSE

    c) return WinDefWindowProc()

    d) break

18. The WM_CREATE message is sent to your window

    a) after it has been created

    b) before it is created

    c) as it is being created

    d) after it has been made visible

19. To destroy a window, you should

    a) Send it a WM_DESTROY message

    b) Post it a WM_DESTROY message

    c) Post it a WM_QUIT message

    d) Call WinDestroyWindow()

20. Every frame window on your screen has a separate window procedure

    a) True

    b) False

Basic Window Concepts and Details

1. The function that creates a pushbutton is:

   a) WinCreatePushButton()

   b) WinCreateButton() with style BS_PUSHBUTTON

   c) WinCreateWindow() with class WC_BUTTON

   d) HWND hwndButton = new button;

2. Inside a window procedure, the variable hwnd by convention refers to:

   a) The client window

   b) The frame window

   c) This window

   d) The system window

3. A window handle is a constant, #defined by the programmer in a header file.

   a) True

   b) False

4. A window ID is

   a) A constant, #defined by the programmer in a header file

   b) A 32-bit variable, usually a ULONG

   c) Returned by the WinCreateWindow() API

   d) What Bill Gates carries to prove he's over 21.

5. Given a window handle and the ID of one of its children, I can find the child's handle using:

   a) WinQueryWindow()

   b) WinWindowFromID()

   c) WinQueryWindowPtr()

   d) WinQueryWindowUShort()

6. When a window is created, its size is by default:

   a) the same as the desktop

   b) half the desktop height and width

c) zero, zero

d) 640 x 480

7. Mouse and keyboard input is initially fed to:

a) The application message queue

b) The system message queue

c) The PM dispatcher

d) The application window procedure

8. To get the first short out of the first MPARAM in a message, the macro would be

a) MRFROM2SHORT(s1, s2)

b) SHORT1FROMMP(mp2)

c) PVOIDFROMMP(mp1)

d) SHORT1FROMMP(mp1)

9. The WM_COMMAND message specifies the ID of the menuitem chosen in:

a) SHORT1FROMMP(mp1)

b) SHORT2FROMMP(mp1)

c) SHORT1FROMMP(mp2)

d) SHORT2FROMMP(mp2)

10. The WM_CONTROL message specifies the ID of the control which generated it in:

a) SHORT1FROMMP(mp1)

b) SHORT2FROMMP(mp1)

c) SHORT1FROMMP(mp2)

d) SHORT2FROMMP(mp2)

Menus

1. The message sent by a menu when a user chooses a menuitem is:

   a) WM_CONTROL

   b) WM_INITMENU

   c) WM_DISMISSMENU

   d) WM_COMMAND

2. The WM_COMMAND message mp1 parameter contains:

   a) A pointer to the menu text string

   b) The menuitem index

   c) The menuitem ID

   d) The menuitem window handle

3. The tool which compiles menus is called:

   a) The Resource Compiler

   b) The Menu Compiler

   c) The Chef

   d) The C Compiler

4. When the user selects a submenu, it will send a WM_COMMAND message with mp1 set to its ID.

   a) True

   b) False

5. Menu item attributes can include:

   a) Checked, disabled, highlighted

   b) Checked, bitmap, highlighted

   c) Checked, disabled, ownerdraw

   d) Disabled, ownerdraw, highlighted

Window Words

1. Because window procedures are called repeatedly, they should use static variables to preserve settings.

   a) True

   b) False

2. The WinRegisterClass() API allows you to specify:

   a) The address of a structure which will contain instance data

   b) A pointer to a function which will set and query instance data

   c) A number of bytes to be reserved for instance data

   d) A number of bytes to be reserved for a pointer to instance data

   e) a and b

   f) c and d

3. The best place to call malloc() in order to reserve memory for instance data is

   a) The WM_CREATE message processing

   b) Before calling WinCreate(Std)Window()

   c) In the WM_ALLOCMEM message processing

   d) After calling DosAllocMem() to obtain a heap object

4. After obtaining a pointer to an instance data structure by calling malloc(), it should be stored in the window words by calling

   a) WinSetWindowULong(hwnd, 0, p);

   b) WinSetWindowWord(hwnd, 0, p);

   c) WinSetWindowPtr(hwnd, 0, p);

   d) WinSetInstanceDataPtr(hwnd, 0, p);

5. Whenever the winproc is called, the pointer to instance data must be reset:

   a) Never; it's already set to the correct value.

   b) At the top of the winproc before any messages are processed.

   c) At the top of each message stub that accesses the instance data

   d) Either b or c

OS/2 Control Program API

1. All the OS/2 Control Program API names start with Os2.

   a) True

   b) False

2. Which calling convention is used by the OS/2 Warp Control Program API's?

   a) C

   b) Pascal

   c) _System

   d) FORTRAN

3. The DosOpen() API returns a file, pipe, or device handle.

   a) True

   b) False

4. The first parameter passed to the DosCreateEventSem() API is:

   a) The address of a HEV

   b) A PPHEV

   c) An error indicator

   d) A HEV

5. The Visual Age C++ compiler can be used to create Family API programs.

   a) True

   b) False

Dynamic Link Libraries

1. A dynamic link library runs in a separate process from its clients.

   a) True

   b) False

2. A single-threaded dynamic link library is compiled using which option?

   a) /Gt+

   b) /Ge+

   c) /Gt-

   d) /Ge-

3. What .DEF file statement tells the linker to generate a .DLL file?

   a) EXPORTS

   b) SEGMENT

   c) LIBRARY

   d) NAME

4. What .DEF file statement tells the linker to generate a .EXE file?

   a) EXPORTS

   b) SEGMENT

   c) LIBRARY

   d) NAME

5. The only entry point into a PM program's .EXE file is the main() function.

   a) True

   b) False

6. A single-threaded client program which uses a .DLL is compiled using which option?

   a) /Gt+

   b) /Ge+

   c) /Gt-

   d) /Ge-

7. Which option does NOT mark a function as exported froma DLL?

    a) #pragma(linkage, external) in the C source.

    b) #pragma(export) in the C source

    c) _Export declarator on the function definition in the C source

    d) EXPORTS functionname in the .DEF file file

8. Dynamic link library files should be placed:

    a) In the same directory as the calling .EXE

    b) In a directory that is somewhere on the PATH

    c) In a directory that is somewhere on the LIBPATH

    d) In the \OS2\DLL directory

9. An “Unresolved external reference” message when linking a program indicates:

    a) You need to specify the DLL’s .DEF file on the command line to link the program

    b) You forgot to specify EXPENTRY on the client window procedure

    c) You forgot to link with the DLL’s import library

    d) There is no IMPORTS section in the DLL’s .DEF file

10. It is possible to create a .DLL that contains no code and no data.

    a) True

    b) False

Processes and Threads

1. The API that starts a child process is:

   a) exec()

   b) DosExecPgm()

   c) DosCreateThread()

   d) _beginprocess()

2. The highest priority thread class is:

   a) Urgent class

   b) Time-critical class

   c) Top-priority class

   d) Mission-critical class

3. Which of the following can DosSetPriority() NOT vary?

   a) Priority of all threads in a child process

   b) Priority of one thread in a child process

   c) Priority of all threads in the current process

   d) Priority of one thread in the current process

4. Which of the following resources can a thread own?

   a) Pipes

   b) Files

   c) Queues

   d) None of the above

5. The minimum and maximum priorities in a priority class are:

   a) -31 to +31

   b) -15 to + 16

   c) 0 to 31

   d) 1 to 32

6. To which class does the system apply dynamic priority variation?

   a) Time-critical class

   b) Urgent class

   c) Idle-time class

   d) Regular class

7. For how long can a winproc tie up a thread before being considered ill-behaved?

   a) One fiftieth of a second

   b) One tenth of a second

   c) One second

   d) Five seconds

8. Which of the following API’s can a non-message-queue thread NOT call?

   a) WinBeginPaint()

   b) WinSendMsg()

   c) GpiDrawText()

   d) WinPostMsg()

9. You are writing a text-mode application. Which API do you use to kick off threads?

   a) DosStartThread()

   b) DosCreateThread()

   c) _beginthread()

   d) _startthread()

Interprocess Communications

1. Name three types of OS/2 2.x semaphore:

    a) RAM, safe and fast-safe

    b) Event, mutex and muxwait

    c) Global, private and protected

    d) Synchronous, asynchronous and fast-safe

2. Which of the following is correct?

    a) Named shared memory can extend across a network

    b) Named pipes can extend across a network

    c) Named shared memory can link OS/2 and DOS applications

    d) A named pipe can link two DOS applications

3. Which option does NOT apply to semaphores:

    a) Named or unnamed

    b) Protected or unprotected

    c) Private or system

    d) None of the above

4. Which of the following statements is NOT true?

    a) Queues can be read in FIFO sequence

    b) Queue elements can be up to 64 KBytes in size

    c) Queues can be read in LIFO sequence

    d) Only the creator of a queue can read it

5. Which modes does a named pipe support?

    a) Byte mode and message mode

    b) Text mode and packet mode

    c) Asynchronous mode and synchronous mode

    d) Protected mode and real mode

6. Which API is used to pass PM messages to another process?

    a) WinSendMsg()

b) WinPostMsg()

c) WinDispatchMsg()

d) WinPeekMsg()

7. User-defined messages must be greater than or equal to:

a) 8192

b) UM_FIRST

c) WM_USER

d) 65536

8. In order to protect access to a presentation space, you would use a:

a) Mutex semaphore

b) Event semaphore

c) WinSetSem() API call

d) Object window

9. How can you stop your character/kernel application from being nuked by the user pressing Ctrl-Break?

a) Subclass the frame window and intercept WM_CHAR messages

b) Install a device monitor on the keyboard and filter out Ctrl-Break

c) Call WinCancelShutdown() before creating your message queue

d) Register an exception handler for the XCPT_SIGNAL_KILLPROC exception

10. How many named pipes can have the same name in a pool?

a) 4

b) 8

c) 16

d) 32

Dialog Windows

1. Which API displays an application-modal dialog window from application resources?

   a) WinLoadDlg()

   b) WinCreateDlg()

   c) WinDlgBox()

   d) WinDisplayDlg()

2. Which API displays a modeless dialog window from application resources?

   a) WinLoadDlg()

   b) WinCreateDlg()

   c) WinDlgBox()

   d) WinDisplayDlg()

3. Which API makes a modeless dialog modal?

   a) WinProcessDlg()

   b) WinSetDlgModal()

   c) WinDlgModal(hwndDlg, TRUE)

   d) WinDlgModal(hwndDlg, FALSE)

4. What is the default message processing in a dialog window procedure?

   a) return WinDefWindowProc(hwnd, msg, mp1, mp2);

   b) return (MRESULT) FALSE;

   c) return WinDefDlgProc(hwnd, msg, mp1, mp2);

   d) return WinDismissDlg(hwnd, DID_OK);

5. Which message allows you to initialize the control windows on your dialog?

   a) WM_SETCONTROLS

   b) WM_CREATE

   c) WM_SIZE

   d) WM_INITDLG

6. In a .RC file, a dialog window is created by which statement?

a) WINDOW

b) DIALOG

c) CONTROL

d) DLGTEMPLATE

7. Which resource compiler statement allows you to set the load options for a dialog?

a) WINDOW

b) DIALOG

c) CONTROL

d) DLGTEMPLATE

8. Which window is usually the parent of a dialog window?

a) The application client window

b) The application frame window

c) The desktop

d) FID_DIALOG

9. Which API runs the font selection dialog?

a) GpiQueryFonts()

b) WinQueryFonts()

c) WinFileDlg()

d) WinFontDlg()

10. Which combination of flags is invalid for a file save dialog?

a) FDS_CENTER | FDS_SAVEAS_DIALOG

b) FDS_HELPBUTTON | FDS_SAVEAS_DIALOG

c) FDS_SAVEAS_DIALOG | FDS_MULTIPLESEL

d) FDS_CUSTOM | FDS_MULTIPLESEL

Index