Introduction to OS/2 Warp Programming

Instructor's 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 specializing 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 guide is intended for the instructor of the course and should not be printed out to the students. This document will help the instructor to prepare for the course and help the students while they are learning the content of this course.

Day 1

Session 1

The first session is intended to provide for a warm-up and provide some background information for the student. The design of OS/2 reflects the design of the 80286 processor for which it was originally written, even though OS/2 2.X runs on the 386 and above and the application programmer is not aware of the underlying segmented memory model.

Slide 2 - About the Course

Explain that the course manual has the slides at the back, and that the students should track the course using these and write any additional notes they want to make on the slides or the facing blank pages.

Questions are welcome at any time - "I use the slides to keep me on track so am happy to answer questions at any time. However, it might happen that somebody asks a particularly arcane or obscure question, in which case I reserve the right to return the seminar to the main topic under discussion. However, I will answer questions later, at my standard consulting rates - a beer for the simple questions and a gin and tonic for the complex ones. If you're very unlucky, you might hit a double malt whisky question!"

NB - I once got into trouble, teaching a course with a large group from the Australian Taxation Office present. They asked a lot of beer questions!

Explain the nature of the lab exercises and how to search for the *LAB* comments, as well as the fact that the size of the blank space relates to the number of lines of code to be written.

Slide 3 - Course Overview

Outline the course topics and give some indication of the time to be spent on them, e.g. "and then on Thurday morning we'll look at advanced PM programming techniques, starting with window words"

Slide 4 - OS/2 is Big

Contrast the size and complexity of OS/2 vs DOS. E.g. OS/2 1.X SDK: 35 volumes of documentation, 115 disks, 75 lbs in weight. Good opportunity for jokes, esp. Tim Patterson's original writing of DOS in one month, hence the name QDOS (Quick and Dirty Operating System), just look at EDLIN - a half-day effort, etc. I often talk about Jolt Cola at this point.

The key point is that OS/2 runs on the 286/386 processors in protected mode, and that therefore applications are essentially impotent - they can compute, but they can't affect the outside world, and therefore rely on the operating system for all I/O, screen painting, mouse input etc. Hence the complexity of the OS.

Slide 6 - 80x86 Real Mode

Discuss address construction, and the limitations on segment size particularly.

Slide 7 - 80286 Protected Mode

Point out that the segment registers no longer contain the base address of segments, but instead contain selectors which are indexes into descriptor tables. Each descriptor contains the actual base address, limit, privilege level, protection info and segment not present flag. What does this buy us? :

Dynamic relocation of segments

More reliable multitasking - in real mode if two programs have the same addresses, they really do address the same memory cells, overwriting each other's code or data and causing a crash. In protmode, the same addresses are the same selectors and offsets, looking up corresponding entries in two different local descriptor tables, which will give two different segment base addresses and two different resulting physical addresses. In other words, programs deal with virtual addresses which the processor will translate into different physical addresses. If trying to 'sell' OS/2, mention that Windows puts everything into a single LDT, allowing Windows programs to party on each others' memory. Tsk, tsk.

More reliable code - can't jump beyond the end of a segment (offset > limit), can't execute data, can't write to a code segment, etc. Also much more proof against virii.

Virtual memory - explain how segments can be swapped in, and how discardable segments get blown away. Explain LRU algorithm and use of swap file. I often explain virtual memory by analogy to a juggler.

Slide 8 - 80286 Protected Mode Benefits

Used to reinforce the previous slide

Slide 9 - Ring-Based Architecture

Explain the role of the descriptor privilege level when performing FAR CALLs and accessing data. Also cover the role of ring two, IOPL and IOPL segments.

Slide 10 - Protected Mode Benefits

Finish up the discussion of protmode with discussion of the maintenance of separate stacks with each ring, and if network operation and software development is important, mention the different philosophies of Novell and Microsoft wrt placement of applications and how each was forced to compromise.

Slide 11 - Intel 80x86 Processor Family

Simple overview of processor capabilities

Slide 12 - The 80386DX Processor

Introduce the 386 and explain differences from the 286. Don't put too much emphasis on the doubling of word size - it doesn't really lead to a doubling of performance, as some may believe. Instead, place the stress on the 4GB segment size, which allows easy porting of flat-address-space programs, and the use of 4 KB pages for swapping.

Slide 13 - 386 Block Diagram

Slide 14 - A 386 Needs Support Chips

Explain the architecture of the typical 386-based machine, showing the need for and operation of cache memory, and the impact of floating-point hardware on certain types of programs.

Slide 15 - A 486 Doesn't

Introduce the 486 as an integrated 386 chipset. Point out the tighter coupling of the modules allows higher performance than the separate chips.

Slide 16 - Intel Processor Performance

The performance of Intel processors from the 4004 onwards can be graphed on log-linear graph paper, with time along the X-axis and performance up the Y-axis. The result is very close to a straight line, with almost no scattering of the points. This includes extrapolation through the Pentium, the P6 and the P7. All Intel has to do is keep shrinking the chip geometries in the same way they have been for over twenty-five years now; there is no fundamental research involved, simply development.

"Does anybody think they will be running DOS on this chip?"

Slide 17 - Applications Cannot Perform Privileged Operations

Apps cannot call the BIOS because a) the BIOS is real mode code, b) the BIOS does not have the performance or features required for OS/2 apps and c) the BIOS lives at a real address while OS/2 apps deal with virtual address (don't complicate matters with PhysToVirt()!).

At first sight, inability to execute from a data segment and write into a code segment would prohibit two things: self-modifying code and overlays (such as dBASE's .OVL files). However, self-modifying code is basically bad news anyway and best avoided (although OS/2 1.x supports it through code segment aliasing and PM actually uses self-modifying code to implement the GPI). Overlays are not even necessary for OS/2 due to the use of virtual memory - just write a 10 MB .EXE file and run it on a 8 MB machine, relying on swapping; it will be slow, but it will work. However, this is ugly - it makes more sense to keep subsystems on disk until needed (if ever). For this, OS/2 provides a facility that does

everything that overlay files did and a lot more, in the form of dynamic link libraries. These will be the subject of more detailed study later.

Slide 18 - Memory Models

Point out that this is not a major concern for OS/2 2.X application programmers; although the system still views memory in segments each application is totally within one segment. However, it is still important to device driver developers and of course, those developing 16-bit FAPI and OS/2 1.X applications.

A quick sketch on the whiteboard with some large boxes for the code and data segments of a small model application, together with a segment to represent some OS kernel code and an allocated data segment provides a good opportunity to discuss the various models and

Slide 19 - The _near and _far Keywords

Make clear the fact that a near pointer is simply a 16-bit offset within thhe default segment, while a 32-bit pointer is the concatenation of a 16-bit selector and a 16-bit offset within *that* segment.

Slide 20 - Introduction to Tools

Briefly describe the major development tools: the compiler (MS C 6.00 or CSet++), integrated development environment (PWB or WorkFrame/2) and the toolkit. Discuss the function of the major utilities such as the icon, font an dialog editors, as well as the on-line help.

Slide 21 - IBM CSet++ Compiler

Slide 22 - Basic OS/2 Compile

Note that this is, in principle, the same as for DOS.

Slide 23 - Larger OS/2 Compile

Now the source is split into several modules which are separately compiled and linked. Again, the same principle as for DOS and other OS'es. However, note that any assembly language must be assembled using MASM 6.00 or earlier. Microsoft has removed OS/2 support from the latest version. (However, I believe that it is possible to buy the latest version and then obtain a 'down-grade' from MS Tech Support).

Slide 24 - Bound FAPI Application

In this case, the compiler must be MS C 6.00, Watcom or some other tool which can generate 16-bit code. CSet++ generates 32-bit code which cannot run on most DOS machines. The resultant .EXE will run under DOS and also under OS/2, in either an OS/2 protmode session or a 16-bit DOS box or VDM. A little thought will reveal that many of the OS/2 commands must be compiled this way

Slide 25 - Creating and Using a DLL

Point out that this slide will return for detailed examination later. The essential points at this level are that now we have some additional code - functions R and S - which are to be compiled and linked separately to form a DLL, rather than being statically linked into ABC.EXE. Why? Perhaps because R and S form a subsystem which can be used by other programs; perhaps because R and S form a subsystem which is infrequently used, e.g. a print or comms subsystem.

In any case, the linker has to be told to generate a .DLL file rather than a .EXE file (although the two are structurally the same), and it must construct a table of entry points, rather than a single entry point for the _main() function. Rather than a huge number of command-line options, the linker is set up for this by driving it with a module definition (.DEF) file.

A second problem will arise when the main program is linked, for its code will call the functions R and S, yet the linker will complain of unresolved external references to these functions. To stop this, we 'compile' the .DEF file (or the .DLL) with a utility called the Import Librarian, to create a library which does not contain the actual code for R and S (that is in XYZ.DLL), but rather some dummy references which allow the linker to write out the appropriate references in the ABC.EXE file so that the loader can link in XYZ.DLL at load-time.

Slide 20 - A Full PM Application

Point out that it is good programming practice to separate procedural code from error messages, prompts, help and other strings, in order to allow for porting to other languages. The base operating system supports this with message files and the MKMSGF utility. PM allows placement of strings and other resources such as window templates, menus, accelerator tables, icons and dialogs in separate source files called resource compiler scripts. The resource compiler tokenizes these into a binary format and then attaches the resources onto the final .EXE file.

Slide 21 - Project Make Files

Slide 22 - Handles

Key concept here is that handles are run-time entities, not known at edit or compile times. At edit time we know file names, window ID's and the like; we only find the handles that allow us to manipulate these objects at run time. We have to have functions to convert one to the other.

Session 2

Slide 1

Slide 2 - Presentation Manager Features

This slide positions OS/2 PM against competitive systems such as Windows and X-Windows/Motif and the other UNIX front ends.

Slide 3 - Presentation Manager Services

Slide 4 - Key Concepts

Introduce the idea of a session (or screen group, in 1.X terminology) as a group of processes which can fill the screen or write to a virtual screen (buffer) when not visible.

Slide 5 - What is a Window?

Ask this question before flipping to the slide. Chances are someone will answer with "a rectangular area of the screen" or similar. Nice try. Stressing the idea of window as data structure at this point prepares the student for the idea of a window as an object, and hence object windows, later.

Slide 6 - Windows

Use this opportunity to introduce Hungarian notation, if it has not already been covered. Point out that WinCreateWindow() creates a single window, while WinCreateStdWindow() creates a family of windows, two of which we need to know handles for (frame and client).

Ask the students how a single C function could return two values at once. You may need to ask how C passes parameters - by reference, by value or by name? Explain that C passes parameters by value and demonstrate with a simple example, that a called function only changes its own copy of a parameter, not the original variable.

Now ask how we could pass by reference. Someone will answer, "by passing a pointer". Show how passing the address of a variable allows the called function to change the value of the original variable in the calling function. This is an important concept which students must understand, as it is fundamental to the operation of the Dosxxxx API, as well as to WinCreateStdWindow().

Slide 7 - Getting Window Handles

Remind students that handles are only known at run-time, so we must therefore obtain them indirectly. Make the generalization that it is better to obtain window handles by knowing the relationships amongst the windows, rather than maintaining global variables or even worse, tables of window handles.

Slide 8 - Frame Windows

Parentage and ownership will be covered in the next module of the course

Slide 9 - Presentation Manager Windows

Point out that what the user thinks of as an application window is actually a family of windows from the programmer's point of view. Stress the point that under OS/2, virtually everything the user sees on the screen is a window (unlike in Windows): (push)buttons, entry-fields, menus, MLE's, containers, spinbuttons, and so on. Whenever the programmer wants to create one of these window components, he uses the same function call: WinCreateWindow().

Slide 10 - Specifying Frame Components

Note that this is one of two ways of creating a frame, but they both use frame creation flags in a similar way. Cover the frame creation flags, mentioning FCF_SIZEBORDER, FCF_SHELLPOSITION and FCF_TASKLIST, as well as FCF_STANDARD.

Point out the OS/2 standard technique of starting a structure with a .cb field, and how this allows structures to change between versions of the operating system, yet the API can tell which version of the structure is being passed.

Explain that a module handle is a handle to an open DLL, and that DLL's can contain resources in the same way as .EXE files.

Slide 11 - Creating a Frame Window

Pretty much self-explanatory - point out that the window position and size is zero and window style not visible to allow for later resizing and repositioning before being made visible.

Slide 12 - Creating a Client Window

Point out that the client is a child of the frame.

Slide 13 - WinCreateStdWindow

Slide 14 - A PM Process

Slide 15 - Logic of a PM Process (cont)

Explain that the preemptive nature of the operating system, coupled with much simpler processing of keyboard input compared to mouse input, could lead to events being processed in the wrong order. Also the definition of a PM process (or thread) is that it calls WinCreateMsgQueue().

Slide 16 - Logic of a PM Process (cont)

Slide 17 - Logic of a PM Process (cont)

Slide 18 - Logic of a PM Process (cont)

Explain that messages are initially sent through the application message queue to a thread which reads it and then dispatches it via the event loop

```
while (WinGetMsg())
    WinDispatchMsg();
```

from where it is dispatched to the window procedure.

Slide 19 - What's a Message

This diagram shows the structure of a queue message - mention that the window procedure is actually passed only four parameters when called, but can obtain the timestamp and mouse position via function calls if required (almost never).

Slide 20 - Logic of a PM Process (cont)

Slide 21 - More on Messages

At this point, a diagram showing the system structure can be extremely helpful.

Lab Exercise 1:

At this point, the instructor should read through the source code for \prog21\lab1\lab1.c with the students, relating the code to the various points made above. Lab 1 does not require the student to edit any source code in order to get a working program, but is mainly intended as a test of the correct installation of the development software and the student's ability to correctly set the compile and link options and generate and run a make file.

Session 3

Slide 1

Slide 2 - Window Relationships - Parentage

Parentage defines how windows are painted on the screen. A window is always positioned relative to its parent, and will be (by default) clipped by its parent.

Use this as an opportunity to stress the distinction between window *handles,* which are run-time entities only, and window *ID's,* which are known at compile-time. Introduce the WinWindowFromID() function, and point out that the frame components have pre-defined ID's, including FID_CLIENT, which can be used to find out the window handles.

Slide 3 - Window Relationships - Ownership

Ownership, by contrast, defines which window is expected to implenment logic to deal with an event in a control window. Usually, control windows are placed on a client window, and the client window logic defines their interaction. The client window will therefore be the owner of the control windows. It is quite common for the parent of a window not to be its owner, as the visual appearance of the application is not the same thing as its logic.

Notice also that the frame is not necessarily the owner of the client, nor is the desktop the owner of the frame. After all, the top-level client is the ultimate arbiter in an application - there is no circumstance under which the client says (in effect) "I don't know what to do here, you take care of it".

Slide 4 - The Frame's Children

More detail on the typical frame components. It might be worth mentioning that the frame has the job of sizing and positioning its various components by updating an array of SWP structures whenever the frame is resized. That is the frame's job - to coordinate and resize and move its children.

The last point on the slide is also worth stressing. Students often have difficulty a) accepting that a window is a generally useful object, *apart* from its on-screen appearance, and b) chaging their thinking from decomposition into subroutines to decomposition into windows. You might draw on the whiteboard a parentage tree for an application which comprises a frame, client, toolbar, container and possibly other windows, and point out that the client is not visible as it 'lurks' behind the container and toolbar. Lab 6 will be a good example of this kind of application.

Slide 5 - Control Windows

A brief introduction to the different types of control windows. A couple of types are missing from this slide: WC_STATIC (static text or bitmap) and the volume control, which is a round control, contrary to CUA guidelines, and only found in the multimedia toolkit.

Slide 6 - What's a message?

Included for revision before discussing message parameters.

Slide 7 - Packers and Crackers

These are the macros designed to assemble and pull apart the contents of the MPARAMs. The most frequently used in a windowproc are the ones at top right, the 'FROMMP' ones. These are used to crack apart the two MPARAMs passed to the winproc. At lower right are the packer macros, also used in a winproc, to assemble a message result or MRESULT. These are the 'MRFROM' macros. At top left are the packers used before calling WinSendMsg() or WinPostMsg(), to assemble the MPARAMs, and at lower right are the ones used to pull apart the returned value of WinSendMsg() - the 'FROMMR' macros.

Slide 8 - Notes on Messages

At this point, it's a good idea, if you haven't already, to draw a diagram showing the PMDD.SYS, system queue, PM router, application message queues, event loops, PM dispatcher and window procedures. Now show the path a sent message takes: back from the sending winproc up to the PM dispatcher, and then to the target winproc. How is this done? Given the target window handle, the dispatcher looks up what class the window is (this comes from a table maintained in the PM dispatcher; entries are created from the parameters to WinCreateWindow()). Now, given the class, it looks up the address of the winproc (also from a table in the dispatcher, entries constructed during processing of WinRegisterClass()), and then calls the winproc. WinSendMsg() therefore is an indirect function call, which works by a couple of table lookups.

Two corollaries follow: if the called winproc hangs, your thread never returns - so much for OS/2 inter-process protection, right? - and when it does return, the message has been processed and there can be a returned value.

Then demonstrate that WinPostMsg() just places a message on the application queue and immediately returns. There is thus no possibility of a returned value, except of course, by the recipient WinPost'ing a message in response. The phrase "Use WinPostMsg() when you can wait on processing of the message" is a little unfortunate - a better way of putting it might be, "when you are able to carry on with doing something else while the message is processed".

WM_QUIT *must* be posted, since it must be in the message queue in order for WinGetMsg() to read it, return FALSE and break out of the event loop while() condition.

The last point about post a message which contains a pointer to memory which will be freed by the time the recipient gets the message is a good reminder that PM is a multitasking system and that programs do not execute from top to bottom.

Slide 9 - Messages

Slide 10 - Useful Messages

WM_CREATE - stress that this is sent to a window *as it is being created,* not before or after. This will be a source of difficulty for most students later in the labs. For example, they will try to refer to hwndFrame in the WM_CREATE processing - but at that time, the WinCreateStdWindow() function has not completed and has not returned a value for hwndFrame! Let them stumble across this for themselves, as they will learn it a lot better that way.

Point out that the control data structure is specific to each window class and should be looked up in the on-line docs, or defined for your own window classes.

WM_INITDLG - Point out that a dialog window is in fact a high-performance frame window, which is subclassed after creation in order to provide additional functionality (e.g. processing of tab keys). Because it is subclassed *after* creation, your dialog procedure has missed the WM_CREATE message, so the system is defined to send it WM_INITDLG instead.

Slide 11 - More Useful Messages

WM_PAINT - Point out that WM_PAINT messages have no parameters. This leads to a discussion: how do you know what/where to paint. Introduce the concept of the invalidated region as the accumulation of invalidated rectangles, and the automatic provision of an hps and rectl by WinBeginPaint() (and also the validation of the region). This might lead to a discussion of the operation of the CS_SYNCPAINT class style, and the fact that without that style, the invalid region is allowed to accumulate until such time as the application queue is empty, at which point the WM_PAINT is generated, allowing the screen to lag behind the user.

This, in turn, could lead to a whiteboard discussion of the PM painting model, in which for many applications, the only place painting is done is in the WM_PAINT processing. User actions will update an internal representation of the application/window state, then a WinInvalidateRect() function call causes window invalidation and consequent generation of the WM_PAINT message. This style of programming is seen in all the labs. The point bears repeating, as it is new to most students, and will be referred to again in connection with multithreading and creation of a separate thread for window painting.

WM_COMMAND - point out that this is the basic message for most logic processing, and the processing breaks down into another switch() statement, based on SHORT1FROMMP(mp1). the fsSource and fPointer fields are rarely referred to.

Slide 12 - More Useful Messages

WM_CLOSE - point out the distinction between WM_CLOSE and WM_QUIT - WM_CLOSE is used to give the user an opportunity to perhaps save work before closing a window (note, not the application) while WM_QUIT is used to terminate the event loop. WM_CLOSE might close a single window, WM_QUIT will close all of them.

Slide 13 - More Useful Messages

WM_CONTROL - the other key message. Point out that SHORT1FROMMP(mp1) is the ID of the control window which generated the message, while the event itself is signalled by the Notify Code in SHORT2FROMMP(mp1). Give some examples, e.g. EN_KILLFOCUS when the user tabs off an entryfield, etc.

Slide 14 - Menus

This is a basic introduction to menus - advanced menu processing (status lines, coloured menus, popup menus) is in the advanced course.

Explain that submenus are actually trees of windows, which are children of HWND_OBJECT and not visible most of the time, but when the user selects a submenu, it is made a child of HWND_DESKTOP and becomes visible.

Slide 15 - A Typical Short Menu

Read through this resource compiler script, pointing out the symbolic constants, use of the tilde (~) to underline a mnemonic letter and the use of the '\t' to space out accelerators on the menu.

Lab Exercise 2 -

This lab exemplifies the model of window painting that was described above. There is nothing terribly tricky about it. You will have to suggest the use of the WinLoadString() function and point out the use of the module handle parameter, as the on-line documentation is basically misleading.

Day 2

Session 1

Slide 1 - Control Windows (and Window Controls)

Says it all, I guess. These are windows which are intended to appear 'on top of' other windows for user interaction.

Slide 2 - Control Windows

The various window classes listed here have their window procedure code stored in and are implemented by the various system DLL's, such as PMWIN.DLL.

These are just the main classes - not shown are classes less commonly dealt with such as WM_MENU and WM_TITLEBAR. In any case, those window classes are not usually thought of as control windows because they don't usually send WM_CONTROL messages.

However, also not shown is WC_STATIC, the static text/bitmap 'control'. I keep forgetting to add it, and it isn't dealt with anywhere else. . .

Slide 3 - Scrollbars

This harks back to functions introduced in the previous presentation, particularly WinWindowFromID() and - though it's not mentioned on the slide - WinQueryWindow() (especially with QW_PARENT). Students should be familar with these functions before tackling Lab 3. You might also mention that Lab 3 is mainly about scrollbars, which should wake them up!

Scroll bars can be created with WinCreateWindow(), and here we are stressing the idea that everything is a window. How do you create a pushbutton? WinCreateWindow(). How do you create a container? WinCreateWindow() and so on. However, it's probably worth mentioning that most windows get scrollbars by using the FCF_VERTSCROLL and FCF_HORZSCROLL frame creation flags on WinCreateStdWindow().

Most 'programming' of scrollbars is done by sending messages, as shown here.

Slide 4 - Scrollbar Notification Messages

These are the messages sent from a scrollbar when something happens that it wants to notify to its owner (notice - not its parent). The SB_SLIDERTRACK notification code is sent when the user picks up the slider and drags it around. Applications should process this message only if they can update the window contents quickly enough. Otherwise, they should pick up the SB_SLIDERPOSITION notification code which is sent when the user finally 'drops' the slider. Similarly, if the user holds down the mouse button over the scrollbar shaft or an arrow, the application may not be able to process the WM_xSCROLL messages quickly enough, in which case the programmer should deal with the SB_ENDSCROLL message which is sent when the user releases the mousebutton.

Notice also that the scrollbar position can be extracted as short 1 from mp2 - there is no need to send an SBM_QUERYPOS message. However, the scrollbar does not automatically repaint itself in the updated position, and so its position should be updated periodically with a SBM_SETPOS message.

Slide 4 - Entry-Field Controls

Although, as this slide states, entry-fields are mostly used in dialogs and are created from the dialog template ENTRYFIELD statement, they can also be created on an application main (client) window. How is this done? Chorus: WinCreateWindow()!

The last point, that an entryfield defaults to 32 characters in capacity, this can be overridden by filling in and passing an ENTRYFDATA structure, as shown in the file \PROG21\FORMS\PMFORMS.C. This structure allows the programmer to specify the capacity of the entry-field, as well as the minimum and maximum numbers of characters which can be selected. Most, if not all, control windows can be passed a similar creation structure ('createstruct') and programmers should check these out before assuming that a window won't do what is required.

The entry-field control has a number of important notification codes, passed as SHORT2FROMMP(mp1) in WM_CONTROL messages. The most important are EN_CHANGE, which indicates that the user has changed the content of the entry-field, and EN_KILLFOCUS, which indicates that the user has tabbed off the entry-field or clicked the mouse elsewhere and is the programmer's opportunity to read the text from the control with the WinQueryWindowText() function call.

Slide 6 - Multiple Line Entry (MLE) Fields

The MLE is a very powerful control, providing something very near to a word processor with a single line of code - WinCreateWindow(hwnd, WC_MLE, ...). It has extensive capabilities, as this slide show, but notice the MLS_READONLY style and warn students to avoid it. Why? Anyone who uses 250KB of memory, just to display some text in a scrollable window is not being professional - just a few lines of code around WinDrawText() can do the job.

Session 2

Slide 1 - OS/2 Kernel Features

Title Slide

Slide 2 - Kernel Features

Explain that the kernel is the base operating system, or control program. Historically, the control program started life at Microsoft as DOS 4.0 (then 5.0, then 286-DOS, then CP-DOS and NewDOS, before becoming OS/2) and its design reflects that. In some ways, it could be thought of as 'DOS on steriods'.

It is for this historical reason that the various OS/2 kernel API's are prefixed Dos-, e.g. DosOpen(), DosFindFirst(), DosBeep(), DosSleep(), etc.

Slide 3 - Applications Programming Interface

Most programmers with some PC experience are familiar with the DOS API. To call a DOS function, the programmer must look up the appropriate function and sub-function numbers in the documentation, and write code which passes parameters either in DX or ES:BX, thus

```
MOV AH, 09H
MOV DX, BYTE OFFSET string
INT 21H
```

and this would (e.g.) print the message at location string.

There are problems with this approach:

1) It involves overhead compared to just calling the desired function directly

2) It cannot be invoked directly from a high level language (e.g. C must call an assembler function intdos())

Slide 4 - The OS/2 API

A further problem with the above approach is that it won't work under OS/2 since INT instructions are not permitted in protected mode ring 3. Hence the OS/2 API is based upon the CALL instruction.

The second point on the slide (works in real and protected mode) mentions that you cannot access DOS through a CALL - however, this a secondary problem which can be resolved through the use of a layer of binding code which will translate into the appropriate MOV AX,xxH / INT 21H sequences. This is precisely how the 16-bit Family API bound executables work.

A secondary problem is this: assume there is an OS/2 kernel function, DosBeep(), which will beep the speaker. Ask the audience: What is its address?

Answer: It depends. It depends on the version of OS/2, the combination of device drivers loaded and the current wind direction (just kidding).

Hence the need for dynamic linking. We don't know in advance what the address of a kernel routine will be, so we must link to it dynamically, at load time or even later, at run time. Dynamic linking is a later topic.

Stress the point that the returned value of a DosXXX() API call is not the desired value (e.g.a file handle, or pointer to an allocated block of memory). Rather it is either zero upon success or an error code on failure. This is very consistent throughout the entire DosXXX() API.

Day 3

Session 1

Slide 1 - Threads - Or How To Walk Down the Street and Chew Gum At The Same Time

Title slide - pretty much self explanatory - allusion to the fact that unlike Windows programs, OS/2 programs can do two things at once, e.g. print in the background. No more waiting for that finely crafted hourglass pointer<g>.

Slide 2 - Multitasking Concepts

The first concept here is that a program is simply a static entity on disk. In the case of DOS, things are easy: it's simply a .COM or .EXE file. In the case of OS/2, it's a little more complicated, since the program can also include code in .DLL files, and those may in turn be shared with other programs. But the basic idea is that a program is a lifeless file on disk.

Load it into memory, perform various address fixups, create a process descriptor and set it running, and you have a process. In simple operating systems, thhe process descriptor is simply a structure where the dispatcher can stick the register contents for the process when it is not actually on the CPU. So all we have to do is create an initial process descriptor with zeroed-out registers (especially IP set to 00000000H) and then 'resume' the process.

To keep things simple, assume a primitive multitasking microcomputer, such as we used to have in the days of MP/M: a box with some 10" x 5" S-100 cards, a separate box with a pair of 8" floppies, a third box with the hard disk (10MB<g>) and external terminals. (This avoids dealing with the complication of how multiple processes share one screen and keyboard). I could walk up to the first terminal, and at the C> prompt, type "WS" (to start WordStar), and the system would search for the file WS.EXE, pull it apart, allocate and load the various code and data segments, set up a process descriptor, and start the process running. WordStar would burst into life on the screen.

If I then walk over to the second terminal, and type "WS", the system could simply do it all over again: load the code and data segments, set up the process descriptor, and so on. But that's wasteful: the two sets of code segments must be identical, since they are loaded from the same .EXE file and don't change while it is running. So why not share the code segments in memory, and save some space? Of course, now at process termination we can't simply blow the code and data segments away, since a second process might be using them, so we have to maintain a usage count for at least the code segments and only free them when the usage count reaches zero.

Slide 3 - Internal Multitasking

I deliberately chose WordStar as an example a few moments ago because it exemplifies a category of programs that perform polling of the keyboard. That was the way things were done back in the days of CP/M, and a lot of DOS programs still do it today. You might ask the audience if they think that is a good design for programs that run under OS/2. The answer, clearly, is no, since such processes continually

loop around, waiting for input, using up CPU cycles and degrading the performance of other processes that are doing useful work.

However, there is a reason why some programs are designed that way, at least for CP/M and DOS. Way back in 1979, WordStar could do what most Windows word processors still can't do today: print in the background. You could be typing away, and realise that you had forgotten to print a document. No problem, just press ^KP, and a little box appears (drawn with +, - and |) which prompts for the filename, press ESCape and off it goes.

Now, by no stretch of the imagination could CP/M 2.2 be described as a multitasking operating system, so it clearly wasn't using operating system features to achieve this. Instead, it was done by breaking open the polling loop, as shown in the slide, and inserting an extra step: "Get a character from the print file and print it'.

Now there's a double loop. I usually animate this for the audience, waving my arms about as I step through the logic. And I point out that actually, it's more complex than shown on the slide:

> "Is there a character from the keyboard?"
>
> "No"
>
> "Is the parallel port ready yet?"
>
> "Yes"
>
> "Get a character from file, then send it to the parallel port"
>
> "Is there a character from the keyboard?"
>
> "No"
>
> "Is the parallel port ready yet?"
>
> "No"
>
> "Is there a character from the keyboard?"

And so on.

Of course, we actually want the program to stop, or block, when there is no input from the user, but if we do that, then it can't continue to print

How would OS/2 handle the problem better?

Slide 4 - Flow of Execution - Subroutines

Of course, we are all very familiar with the concept of subroutines - so much so that we take it for granted , or use them subconsciously.

As the slide shows, the flow of control proceeds through the main routine until it comes to the subroutine call, at which point it pushes the address of the next instruction - the return address - onto the stack and then jumps to the first instruction of the subroutine. The subroutine then runs until the CPU encounters a RETURN instruction, at which point it pops the return address of the stack and resumes execution.

We mostly visualise this as shown on the left of the diagram. Of course, what actually happens in terms of elapsed time is on the right of the diagram: the main routine is suspended while the subroutine runs, then the main routine resumes.

It would be kind of neat if we could have the subroutine run while the main routine continues, and OS/2 permits this, as shown in the next slide:

Slide 5 - Flow of Execution - Threads

Rather than calling the subroutine, we instead call the OS/2 kernel function DosCreateThread(), passing as a parameter the address (in C, name) of the 'subroutine' function. The 'subroutine' now starts to execute, while the main routine continues.

Of course, both routines cannot really execute at the same time on the same processor - the CPU can only do one thing at a time. In reality, the operating system is timeslicing back and forth between the two threads so quickly that the user (or programmer) gets the illusion that everything is happening simultaneously, in the same way as a movie picture is really a succession of still frames, but we get the illusion of movement.

Although the programmer knows that things really happen one step at a time, he must never attempt to write programs that depend on this knowledge, for two reasons:

First: in the absence of other mechanisms to enforce serialization (e.g. semaphores), the programmer has no control over the sequence of execution of instructions. The operating system preemptively timeslices whenever it needs to, and this is not under programmer control. Indeed, it will be different almost every time.

Secondly, on some systems (those that run OS/2 SMP) things really do happen at the same time, on multiple processors. Indeed, this is one reason why Intel are putting multiple execution units into future X86 processors.

Slide 6 - Processes vs Threads

There are some important definitions and concepts on this slide.

First: A process is the instance of program execution. It is the OS/2 unit of resource ownership. In other words processes own things, not threads. First question to the audience: What resources might a process own? Answers you are looking for include: memory segments/objects, files, semaphores, pipes, queues, windows on the screen - in fact, almost anything for which there is an API for creation or allocation. Processes do not own threads, however: threads are just part of the process, in the same way as you don't own your legs - they are part of you and you couldn't run without them.

A process always comprises at least one thread - it couldn't run otherwise. And although resources are allocated or created by virtue of a thread making an API call, the resulting resource belongs to the process of which the thread is a part - not the the thread itself. Threads don't own things, other than their priority and status, stored in a thread descriptor which replaces part of the functionality of the process descriptor.

Second: A thread is a dispatchable entirty. It is the OS/2 unit of execution. In other words, when the system timeslices, it is timeslicing threads, not processes. A system could be running two processes, one comprised of one thread, the other comprised of nine. If all else was equal (which it never is), out of every ten time-slices, the first process would get one, while the second would get nine.

Now, before you think "I know! I can make my process run blindingly fast by dividing the work up amongst lots of threads!" let me issue a warning. It will still take about as long to complete many tasks, but the user will notice that your application is killing the things they have running in the background (users are surprisingly sophisticated about these things - last week they didn't know what a mouse was, this week they're critiquing your program design). They know that although everything else is running slowly, it is your program that is causing the problem, and they will drop it if they can and bad-mouth it to other users. Lotus applications, for example, have taken a lot of this kind of criticism, as have Microsoft's OS/2 applications (though they perform poorly because they are single-threaded, simple-minded ports of Windows applications).

The last point says that an OS/2 process will typically have multiple threads. A better statement might be 'should have multiple threads', or 'a well-designed OS/2 application will have multiple threads'. OS/2 users are critical of single-threaded applications on account of their unresponsiveness. Windows users have no choice, since Windows is purely single-threaded, but then, Windows users aren't critical, or they wouldn't be using Windows, would they?

Slide 7 - Applications for Threads

Pretty much self-explanatory, with the exception of 'Performance Monitoring'. This must be done by comparing the performance of low-priority and regular-priority threads, since OS/2 does not have well-documented API's for extracting, e.g. CPU time vs real time. You could discuss the operation of PULSE.EXE as an example, later when discussing thread priority classes.

Slide 8 - Traditional Terminal Program Design

Ask the audience to identify problems with this design. These include:

Tight coupling of the send and receive sides of the program. A bug in one will bring down the other.

Simplistic design. We wait for a keystroke, then send it without waiting to see if the serial port is ready to accept it.

No provision for expansion of escape sequences. This is going to involve setting flags to indicate we are in an escape sequence, and will have to be interleaved with the send side of the program.

And the big one you are looking for: If the system is quiescent, i.e. not sending anything and not receiving anything, then the program is simply spinning around the main loop, sucking up CPU cycles while not achieving anything useful. You may needto give them a hint here.

Nonetheless, this is the basic design of all terminal programs for single-threaded systems, from the original CP/M XMODEM through to current Windows programs. Now let's see how OS/2 can improve on this:

Slide 9 - OS/2 Terminal Program Design

The OS/2 version of the program has (at least) three threads. The main thread simply kicks off a transmit thread and a receive thread, then processes input such as user commands to set baud rate and parity, upload files, etc. The transmit thread simply calls the OS to get a keystroke, and blocks at that point. It does not return until it has got a keystroke, then sends it. Likewise, the receive thread simply waits for an incoming character, then displays it.

Notice that when nothing is happening on the system, this process is not consuming any CPU cycles, and therefore is not degrading other applications' performance.

There are some other benefits. It now becomes easier to break open the receive thread loop and insert code to handle escape sequences. The transmit and receive sides are now decoupled, helping to isolate bugs.

And we can deal more easily with buffering to deal with slow lines by having a pair of circular buffers - one for transmit and one for receive - with a thread for the input side and a thread for the output side, synchronised by some semaphores. This five-thread design is much more elegant and easier to maintain than a single-threaded system can possibly achieve.

These slides illustrate why OS/2 is such a great operating system for communications and networking applications.

Slide 10 - OS/2 Task Scheduler

The scheduler is preemptive. It's a good idea to contrast this with non-premptive, or cooperative multitasking, as implemented by Windows. As one wag put it: "If your brain was running Windows, you'd have to stop thinking in order to breathe"! Point out that Windows relies on processes voluntarily yielding the CPU, and that a poorly-designed or buggy application can hog the CPU, stopping everything else from running and effectively crashing the system (of course, a really badly-designed program can do the same thing to Presentation Manager in the current incarnation, but it's a lot less likely).

Time Critical Class threads are typically used to respond to urgent situations, e.g. flushing a buffer to disk before it overflows. They spend most of their time blocked, so have minimal impact on lower-priority threads.

Server Class - also called Fixed-High Priority Class - is used for background and daemon threads on a user workstation that must have a higher priority than the user's own work application threads, e.g. network (e.g. peer server) code, SNA gateways.

Regular Class threads make up the bulk of application code. Their priority is varied by the system - threads in the foreground process get a priority boost so that the application the user is working with can be made more responsive. This is a key design feature and distinguishes OS/2 as a single-user workstation operating system from older systems which are designed to support multiple users on a fair basis. This feature can be disabled by setting PRIORITY=ABSOLUTE in CONFIG.SYS. The system also varies priority based on other criteria (see Deitel and Kogan for more details). For example, if a thread is doing a lot of I/O calls but mostly blocked waiting for device status to change, then it will receive a priority boost since that will drive a peripheral much harder but have negligible impact on CPU usage.

If a regular class thread is denied the CPU for more than three seconds (by default) by virtue of other regular class threads receiving a boost, then it will receive a boost to the top of the class for one time-slice, allowing it to run briefly, then will be dropped again. This is used to prevent deadlocks and allow background tasks to at least crawl towards completion. This time period can be overridden using the MAXWAIT= parameter in CONFIG.SYS.

Idle Time Class threads are used for low-priority tasks, such as background diagnostics, batch tasks, generating Mandelbrot sets, etc.

Slide 11 - Why Threads Are Important

This slide has a typo on it, which makes it less than crystalline in clarity.

As students will recall, PM is a message-passing system (remember that diagram?).

There are multiple application input queues. However, their operation is synchronized, so that a message will not be read off any of the application queues until the previous message has been processed (i.e. winproc has dealt with it and returned, and WinDispatchMsg() has returned). This is necessary in order to support predictable type-ahead operation, e.g. you click on the corner of a window and start typing - the window must be brought to the foreground and given focus so that the keystrokes go to the correct window. So, the original WM_BUTTON1DOWN and all subsequent WM_ACTIVATE and WM_SETFOCUS and other messages must be processed before the WM_CHAR messages get processed, otherwise your keystrokes would go to the wrong window.

This is a design feature, not a bug. It is possible to have multiple asynchronous input queues (a la Windows NT) but that has its own problems and can get real ugly.

So, if a winproc goes off and does something (e.g. involving NETBIOS or APPC) that may take some seconds, the system will not process any more messages until those seconds have elapsed and the winproc returns.

This is BAD.

So: The One-Tenth Second Rule. If processing a message will take more than one-tenth of a second, do the processing in a secondary thread. Allow the main thread of the winproc to return immediately, so that subsequent messages can be processed.

Actually, this started out as the one-tenth second rule, then was amended to the one-second rule, when it was realised that on vintage 1987 286 PC's, everything took more than one-tenth of a second. However, these days, on 486's, we have generally agreed (at the 1993 ColoradOS/2 Conference) that it should be amended to the one-twentieth second rule.

You can give some examples of where this has been used to make OS/2 applications more responsive than their Windows equivalents, e.g. DeScribe for OS/2 vs DeScribe for Windows, Pagemaker, Ami Pro (not a great example, it must be admitted).

Slide 12 - How to Create a Thread (OS/2 1.x)

Included for completeness, not vitally important for most courses. Notice that you can't pass a parameter to the thread function, and that the stack pointer must be initialised to the top of the stack (which of course, suggests a work-around way of passing parameters).

Slide 13 - How to Create a Thread (OS/2 2.x)

This version of the DosCreateThread() API is much more usable. Notice that the thread function can be passed a void pointer, which allows passing of multiple parameters in a structure to which this points (or a single 32-bit parameter can be passed by casting it to a VOID * and back again).

Two bits are defined in the ulThreadFlags parameter. The first allows the thread to be created in a suspended state, and then set running with the DosResumeThread() API. This allows multiple threads to be created and started in a defined sequence (remember, this is a preemptive system). The other bit allows the thread stack to be pre-committed. Normally, only the stack for the primary thread is precommitted (STACKSIZE statement in .DEF files or LINK command line parameter sets this). By default, secondary threads have their stacks automatically committed as required by using a guard page (see D&K for details).

The stack is also automatically allocated by the system, unlike in OS/2 1.x.

Slide 14 - _beginthread

I usually start by asking the audience why this function isn't called DosBeginThread(). Eventually someone twigs that it is not an API, but is actually part of the C compiler function library. Why? Because it performs C run-time initialisation on a per-thread basis (e.g. of certain statics required for error-handling, again on a per-thread basis). For OS/2 1.x programmers, it is also much easier to use than DosCreateThread(), although the 2.x version was reworked to make it more usable.

This then leads to a discussion of the need for multithreaded libraries for OS/2. The basic problem is that reentrant code is a bugger to write and is slower than non-reentrant code which makes extensive use of statics. This is especially true of complex functions such as printf() which is essentially a small language interpreter.

Remember, the process owns the code segments/objects that contain the library functions, and the linker, which sticks them into the .EXE, knows nothing about threads. So it won't duplicate functions on a per-thread basis. So reentrant versions are required for multithreaded programs.

Actually, this is only a problem for programs which make extensive use of C functions for I/O or math. PM programs, especially, don't use printf() or scanf(), and all the OS/2 API's are reentrant, so it's less of an issue. Bear in mind, also, that many of the simpler functions (strcpy() etc.) are reentrant, even in the single-threaded libraries, and are thread-safe in all cases.

One final point about _beginthread() - on the majority of slower (e.g. 386) machines, _beginthread itself takes more than one-tenth of a second, and so programs which use it cannot comply with the one-tenth second rule!

Slide 15 - Thread Functions

Just a brief walk through some of the more interesting thread-related functions.

You might point out that DosSetPriority() varies priorities relative to the current value, rather than setting them absolutely (hence the signed short parameter sChange), and that it deltas all threads in a specified child process simultaneously, so that they maintain the correct relative priorities.

Slide 16 - Thread Types

This slide lays down the guidelines for which threads must obey the one-tenth second rule. Notice that there is an exception to the first point: a message-queue thread that is running an object window is exempt from the one-tenth second rule, since it does not form part of the user-interface subsystem.

I usually like to draw a diagram on the whiteboard at this point, showing the design of a typical client-server application: From left to write I draw boxes labelled

User-interface server (What is this? I ask. No-one gets it. It's PM)

User-interface client (Your app winprocs and event loop processing)

Business logic

DBMS client (generally libraries for accessing a remote database)

DBMS server (may be a server process on the same machine. More likely, a separate server machine)

Under the user-interface client and server I draw a loop and write in it: <1/10s. Under the DBMS client and server I draw a loop and ask what response time we're talking here. It can be several seconds.

Obviously we couldn't have a single loop running through the user interface code and the database client stuff, as the response time of the database means we fail the one-tenth second rule. So how do we get two loops? Two threads, obviously. This is why client-server development pretty much demands multi-threading, is why OS/2 is such a great client-server platform and why Windows is such a pain in the ass.

The business logic can go in either loop. That's another design decision which is quite interesting in itself.

This is a good place for a coffee break.

Slide 17 - Interprocess Communications

This slide just provides an overview of what is to follow.

Slide 18 - Anonymous Pipes

Pretty obvious. You might mention DosQFHandState(), which allows the programmer to tell if a file handle has been redirected. There is more of this in the section on file handling.

Slide 19 - Named Pipes

Named Pipes started life as a network protocol developed by Microsoft for MS LAN Manager. IBM wasn't going to use LAN Manager, but got forced into it when their own SNA LU 6.2 network code proved to be amazingly sloe (subsequently much improved). Although repackaging LAN Manager as IBM LAN Server, IBM still wanted to promote LU 6.2 as the preferred client-server protocol, so they announced they wouldn't document or support Named Pipes. Typically, Microsoft played an end run around the IBM establishment by moving Named Pipes out of LAN Manager into the OS/2 kernel. So IBM was stuck with them. This is why they work locally on a stand-alone machine, as well as between a workstation and server across the LAN.

Of course, after telling us all we ought to use named pipes and forcing IBM to adopt them, Microsoft has subsequently lost interest and dropped them, shifting to a sockets/RPC model. IBM in typically stoic style, is making a virtue out of loyally supporting a feature they never wanted in the first place, but that their customers might be using.

Actually, named pipes are neat and simple, though not as good as LU 6.2 for supporting multiple platforms over wider-area networks.

Named pipes can carry either structured data or lines of text, and can be bidirectional. I visualise them (seriously) as something like the voice tubes that used to link the bridge to the engine room on ships.

Slide 20 - Named Pipe Programming

The server logic is pretty obvious, it's just a typical server logic loop. You might need to point out that LAN Server / LAN Manager / NTAS servers are named with two sloshes in front of the machine name, and that C quoted strings use slosh (backslash) as an escape character, hence the requirement to double them when only one is wanted.

Slide 21 - Named Pipe Programming (cont)

This slide shows three different approaches to the logic of a named pipe client. This illustrates the point that OS/2 is a very rich environment, with lots of ways to do things.

Ask the audience what is unique about the first approach. The answer you are looking for is that all the API's are family API calls that are supported in DOS - they are the standard DOS function calls to open, write, read and close files. In other words, to a DOS or OS/2 client program, a named pipe can be viewed as a kind of 'interactive file'. This also means that DOS clients are supported, and that client programs can be written using any language or tool that can access files.

The DOS clients can run either on a DOS network workstation, or in a Virtual DOS Machine under OS/2 2.X. Th latter approach allows existing DOS applications to be integrated, using OS/2 to multitask them, and also allows interprocess communications between OS/2 and DOS applications. (We recently did a project recently for a triathlon sponsored by Borland, who wanted times to be captured and fed to a Paradox database. This was done by using an OS/2 data acquisition program to feed Paradox through a named pipe).

The remaining two techniques cannot be used under DOS without a Microsoft LAN Manager Programmer's Toolkit or IBM LAN Server Programming Tools and Information, as the special named pipe API's require special libraries. However, they are progressively more efficient as far as the network is concerned: The first technique passes twelve packets over a NETBIOS network, the second passes four and the third passes two packets only. Remember, this is network efficiency - application code efficiency may be an over-riding consideration.

Slide 22 - Giveaway Shared Memory (1.x)

Shared memory has been mentioned in an earlier slide - here we are introducing the API's. This slide shows the difference between gettable and giveable shared memory, which also applies in 2.X (and 3.X).

Slide 23 - Giveaway Shared Memory (2.x)

The major difference between the 1.x and 2.x API's is that version 2.x refers to memory objects by 32-bit pointers, while 2.x used 16-bit segment selectors. Other than that, it's pretty much the same.

Slide 24 - Named Shared Memory

Notice that, when using giveaway shared memory, the applications must use some other form of IPC first, in order to pass the selector or pointer between the applications (since selectors and pointers, like handles are run-time allocated entities). A typical technique is passing aPM message.

Named shared memory actually appears in the file-system namespace, allowing applications to previously agree upon a unique name (usually defined in a shared header file). No other IPC mechanism is necessary, which is why named shared memory is used in the IPC lab exercise for this course.

Slide 25 - Queues

These queues should not be confused with the PM message queues, which are a completely separate subsystem.

The reason that queues offer high performance is that they do not usually copy large blocks of data. Instead, the data is placed in shared memory and the queue is used to pass a pointer to it. This avoids the overhead of copying blocks of data onto and off the queue.

The only process that can read a queue is the process that creates it.

Notice the 16-bit and 32-bit queue subsystem incompatibility. This makes it difficult to port applications by testing a new 32-bit client against an old 16-bit server, for example.

Slide 26 - Queue Functions

Self-explanatory, except that for some reason, I omitted DosPeekQueue().

Slide 27 - Semaphores

Should really be Semaphores (1.x). This is the way that 1.x classifies semaphores - by where they are located. RAM semaphores are fast, but easy to over-write. System semaphores are much safer, but slower, and operate between processes.

In OS/2 2.x, almost all semaphores are effectively fast-safe.

Slide 28 - Semaphore Functions

Self-explanatory

Slide 29 - OS/2 2.1 Semaphores

An event semaphore is used to signal the occurrence of an event or quite commonly the completion of some process, such as loading a file, or repainting of a window.

A mutex semaphore is used to protect a serially-reusable resource, such as a file or a presentation space.

A muxwait semaphore (multiplexed wait) is used to combine event or mutex sempahores (can't mix types) to (for example) acquire multiple resources before proceeding or to acquire any one of several resources.

Private semaphores exist in the memory of a process and are prone to overwriting by rogue pointers. Used for inter-thread communications. Shared semaphores are like 1.x system semaphores.

Slide 30 - Event Semaphores

In this slide, the three threads in the right column all open an event semaphore which has previously been reset by the thread on the left. Resetting an event semaphore is like turning a traffic light to red: when the three other threads call DosWaitEventSem() they block, and will not return until the first thread posts the semaphore (turns the light green).

Slide 31 - Mutex Semaphore

In this slide, three threads each in turn, request a mutex semaphore (the first gets it, the others have to wait until the previous thread releases it), do something with the protected resource, and then release the mutex.

Slide 32 - Signals

Signals are not really treated in depth in this course, but they need to be mentioned because they are the primary form of asynchronous communications in OS/2. It doesn't matter what you place in shared memory, pump into a pipe, stuff into a queue or how much you post semaphores - if the receiving process is hung up doing something else, blocked waiting for something else, has crashed and burned or is just choosing to ignore you, there's not a darned thing you can do about it.

But a process cannot ignore a signal. For example, on receiving a SIG_KILLPROCESS signal, a process will suspend thread 1 and use it to run a previously-registered signal handler. This function should terminate other threads, as elegantly as possible, flush all file buffers, close files, release or destroy other resources and then exit. If the process has not registered a signal handler, then the system provides a default one (indicate slitting throat with knife). Your proces is gone. The system will clean up the resources it knows about for that process, but it knows nothing about internal buffers and cannot flush them.

In order to cope with this (especially within DLL's where a client process could crash and burn) use the DosExitList() API to build a list of functions which will be invoked in reverse sequence in the event of the untimely demise of your process.

Slide 33 - File Handling

Title slide

Slide 34 - Opening a File

This slide is a little confusing. Although OS/2 2.x does not support the DosOpen2() API, its version of DosOpen supports the additional parameter (peaop) which is required to support EA's. Lots of parameters on these function calls, all of them explained in the following slides.

Slide 35 - Actions and Attributes

The actions parameter is set upon return from the function call and tells the programmer what happened.

Slide 36 - Open Flags

Self-explanatory.

Slide 37 - Open Mode

The flags marked with an asterisk are those which can be changed dynamically using the DosSetFHandState() API. Those with no asterisk can only be set on the DosOpen() function call.

Note that OPEN_FLAGS_WRITE_THROUGH only disables write-behind buffering within the system. Any caching done by the disk driver can be disabled with OPEN_FLAGS_NO_CACHE.

The locality flags help the HPFS IFS driver to figure out the best place to locate the file in order to minimise fragmentation, as well as cache allocation.

Slide 38 - Access Mode and Slide 39 - Share Mode

These two sets of flags together control how the system will deal with attempts by multiple processes to access a file. A little thought makes all clear<g>.

Slide 40 - Reading and Writing

These two functions are the basic low-level API's. Higher-level functions supplied by the language libraries (e.g. read(), write(), fread(), fwrite()) all wind up calling these functions.

Slide 41 - Moving the File Read/Write Pointer

This function is analogous to the C lseek() function.

Slide 42 - File and Region Locking

File locking has been dealt with in Slides 38 and 39. However, region locking is dealt with by this API. It takes two parameters for the lock region structure pointers, either of which can be NULL. Having two _FILELOCK structures allows the programmer to 'walk' through a file, unlocking the previous region and locking the next.

Slide 43 - Miscellaneous Functions

DosBufReset() is required to flush buffers and force output, e.g. to the screen.

Normally, a child process will inherit the file handles of its parent as part of the environment. However, this causes a problem - what if the child closes a file (handle) while the parent is using it? The file is closed, and attempts by the parent to read or write it will fail.

This problem is solved by using DosDupHandle() to create a duplicate file handle for the same file which the child can use independently and close without affecting the parent.

Day 4

Session 1

Slide 1 - Advanced PM Programming

This session covers some advanced programming concepts, the major one being the use of window words.

Slide 2 - The PM API

This slide is put here, basically because it doesn't fit anywhere else. The OS/2 API is categorised, as shown by the API Prefixes column, into the various subsystems. The Dev prefix relates to device drivers, the Dos prefix is for the kernel API, the Drg prefix is for direct manipulation functions (drag and drop), the Gpi is, of course, the Graphical Programming Interface, the Prf API's are for accessing the INI files (Profile files), the SPl functions relate to the spooler subsystem and the Win API is the window manager which has been our main topic.

The OS/2 2.0 API has been made much more orthogonal and is better organised that the previous versions. Following the model of the Microsoft LAN Manager API, the function names have been compiled from verb/noun pairs, and anomalies have been resolved. For example, to obtain values which the system already knows is always xxxQueryxxxx in 2.X, whereas in 1.x it was sometimes Query and sometimes Get.

Slide 3 - Instance Data

I usually introduce this topic by sketching out a simple winproc on the whiteboard. The example will be the code used in Lab 9: to display a user-selectable quote in user-selected colors. So, borrowing from Lab 2, we have a variable szQuote[MAXSTR], and we add two variables ulFColor and ulBColor. The WM_PAINT processing is modified by adding the color variables into the WinDrawText function call, and some WM_COMMAND stubs are added for case ID_FRED, ID_FGREEN, through to ID_BBLUE, which store the appropriate color constant CLR_RED, _GREEN or _BLUE into the appropriate variable and then invalidate the window to cause WM_PAINT to be resent.

"OK, will this work?", you ask the audience. Silence, then one soul will volunteer that it won't, because if it did, you wouldn't have asked the question. "OK, so why won't it work?" A few of the students will skate around the answer, though you may have to prompt them by asking, "OK - what storage class are the color variables"?

"Right - automatic. And where are automatic variables stored"?

The answer you are looking for is "on the stack". So: when the window is created, the WM_CREATE message gets sent. The system calls the winproc, and on entry the variablea are created on the stack, then the color variables are set to the initial defaults. Then the winproc is exited, the stack rolls up and the color variables disappear again. A little later, the winproc is invoked with a message of WM_PAINT, the winproc is entered, the variables are created on the stack, and the WinDrawText() function uses whatever

is at those locations on the stack as the colour parameters. You might get lucky and have garbage colours, or you might get unlucky and have the correct colours - I say unlucky because the program will display incorrect colours at some point and the fact that it worked correctly at first will mask the real reason.

Later, the user chooses a different color and a WM_COMMAND message is sent, and again the variables are created on entry and destroyed on exit from the winproc. It might appear to work, but generally it won't.

"OK, how do we fix that?"

"Right, make the variables static. Now, they are known only within the winproc, but their values persist outside it. Does that fix the problem? What if there are two windows being displayed?"

Someone will eventually work out that as soon as the user changes either of the colour variables, the next WM_PAINT sent to the other window will cause it to be repainted in the new colors too.

"OK, how do we fix that?"

Now the suggestions will come thick and fast: since the window handle is a parameter to the winproc, you could build a table of structures containing the window handles and colors/quotes for each window, then match the window handle to get the correct structure and hence colors. And other, more imaginative schemes.

Time to backtrack. "What is a window?", you ask. "Remember back to Day 1 Session 2".

"Right, a window is a data structure. Wouldn't it be neat if we could modify that data structure to include our color and quote information?"

The problem with this approach is that in OS/2 (up to and including 2.11) all the window data structures are stored in a single 16-bit data segment, restricted to 64 KB max, and it is not neighbourly to make the window data structures too large. So what we usually do is to allocate memory in our application for the data structure 'behind' the window and then store a pointer to it into the window words.

I then modify the winproc on the whiteboard to look like the solution to Lab 9: define a structure as shown in LAB9.H, together with a macro to extract a pointer to it from the window words, then add code to the WM_CREATE processing to malloc() space for the structure, store the pointer into the window words and then dereference all the variables off a pointer which is extracted using the macro on entry to the winproc. See the solution to LAB9 for the details, although the course notes explain the principles and have similar code (this may change in a future release: the course notes do make Lab 9 rather too easy).

This slide and the ones which follow may be best left until after Lab 9 is completed, and used for revision and amplification of the points brought out during the Lab.

Slide 3 - Instance Data

This slide summarises the previous discussion and the course notes.

Slide 4 - Allocating Window Words

This explains the last parameter to the WinRegisterClass() API which was earlier glossed over or not mentioned at all. A useful point to make is that all predefined control windows in the system have four bytes of window words reserved for use by applications which subclass the window (on which more, later).

Slide 5 - In the Winproc

This slide repeats the detail from the above discussion on allocation of memory for a structure in the WM_CREATE message processing, followed by retrieval of the pointer. The use of selectors (sel) applies to 16-bit code only.

Slide 6 - Object Windows

Another advanced technique is the use of object windows. By now the students should be starting to think of windows as self-contained (encapsulated) objects. You might introduce some material on object-oriented programming, relating creation of windows to the implementation and instantiation of classes and objects in (e.g.) C++, or concepts such as encpasulation and data hiding in older languages.

The only apparent drawback to using windows as a way of implementing classes is that they appear on the screen. Actually, this is not necessarily a drawback, as it can be useful to have objects which are visible for debugging purposes.

However, PM allows the programmer to create windows which are not and never can be visible, by making them children of HWND_OBJECT rather than HWND_DESKTOP. These object windows are generally run in secondary threads, so that the secondary thread looks like this:

```
WinInitialize()

WinCreateMsgQueue()

WinRegisterClass(object window class)

while(WinGetMsg)

   WinDispatchMsg()

WinDestroyWindow()

WinDestryMsgQueue()

WinTerminate()
```

and the winproc looks rather similar to a conventional window procedure, except that it never needs to process WM_PAINT, for example, since it is never sent it. The major system-initiated messages which are sent to an object window are WM_CREATE and WM_DESTROY, and WM_QUIT.

Object windows are usually used to allow secondary threads to be dispatched to perform any of a number of different functions, such as file reading/writing, initiating NETBIOS commands and waiting on results, etc. Because an object window is not part of the user interface component of PM, it is not subject to the one-tenth second rule.

There is an optional exercise in the \PROG21\LAB15 directory which shows how to use an object window. It is in the two files OWS.C (main thread/window code) and OBJECT.C (secondary thread/object window code).

Slide 7 - Subclassing

This slide illustrates the simplest technique for subclassing a window. However it has some restrictions - a more powerful technique is covered in the Advanced PM Programming course.

The major restriction is that the window must be created before it is subclassed (perhaps this technique should be called 'subwindowing' rather than 'subclassing'). As a result, the original window procedure has already received and dealt with the WM_PAINT message before the window is subclassed.

Incidentally, this is the reason why, as will be seen in a couple of slides' time, dialog procedures receive WM_INITDLG rather than WM_CREATE. A dialog is actually a high-performance subclassed frame window, and by the time the subclassing has been done and your dialog procedure is 'registered' as the winproc, the original winproc has already processed the WM_CREATE. So the system sends WM_INITDLG as an opportunity for your winproc to perform initialisation.

There is sample lab exercise which demonstrates subclassing in the \PROG21\LAB15 directory, named SUBCLASS.C. This example subclasses the listbox control to handle a menu. Advanced subclassing techniques are covered in the advanced programming course.

Slide 8 - Dialogs

This slide is pretty much self-explanatory.

Slide 9 - WinDlgBox()

The reason that the resulting dialog window is application modal is that this function invokes its own while(WinGetMsg())/ WinDispatchMsg() event loop code, filtering out just the messages it is interested in.

By convention, the argument passed to WinDismissDlg() is the ID of the button which the user selected in order to dismiss the dialog (e.g. IDPB_OK or IDPB_CANCEL). The WinDlgBox() function returns this to the calling window procedure so it can tell whether to pick up data or not.

Data is passed to a dialog window in the fields of a programmer-defined structure; the only restriction is that the first two bytes of the structure MUST be a USHORT specifying the total size of the structure.

Slide10 - WinLoadDlg()

This function call produces a modeless dialog. The EPM editor provides a good example of a modeless dialog in its search and replace dialog, which remains on the screen through mutiple searches,

Slide 11 - WinCreateDlg()

Not very often used, but does offer the possibility of dialogs created at run-time, e.g. upon logging on to a database and selecting a table, as opposed to the technique used in Lab 3.

Slide 12 - Dialog Templates

Use this opportunity to examine a typical dialog and to demonstrate the Dialog Editor

Day 4

Session 1

Lab

Session 2

Lab

Session 3

Slide 1 - Standard Dialogs

The standard dialogs started life as part of the Microsoft OS/2 1.x SDK. A number of the samples needed code for File Open/Save, and someone decided that this would be a good candidate for encapsulation as a DLL. This was duly done and supplied as an example in the SDK. Later, the same thing was done for Windows and supplied as part of the system, and the same was done for OS/2 2.X.

The standard dialogs handle all the messy details of the file save and replace dialogs and much of font selection for the programmer, and do so in a standard way. In the OS/2 2.1 Advanced Programming Workshop course, we cover font handling and construction of a font menu, but take it from one who has been there, done that and suffered: use the standard font dialog wherever possible!

Slide 2 - FILEDLG Structure

Both the dialogs are driven by rather large and complex structures which are passed to them. In practice, however, most of the elements of the structures are not used and can be simply zero'ed out using memset().

The structure is passed to the dialog window, and in the simplest case, returns withh the szFullFile array filled with the full pathname of the desired file.

Slide 3 - File Dialog Flags

Most of these flags are self-explanatory; for example, FDS_CENTER simply means that the dialog will be centered in its parent window. FDS_CUSTOM means that a custom dialog template will be used, giving an application-unique appearance.

When FDS_FILTERUNION is turned on, the dialog uses the union of the string filter and the EA type filter, rather than the default of the intersection of the two.

Turning on FDS_PRELOAD_VOLINFO causes the dialog to preload the volume label information and current directory for each drive (by default, the volume label information is blank and the current directory is the root).

FDS_ENABLELB allows the user to select from the file list box in a Save As... list box. Default behaviour is to disable the listbox.

The FDS_INCLUDE_EAS flag causes the dialog to always query EA information, even though the extended attribute type filter has not been selected.

FDS_MODELESS is another useful flag which was omitted from the slide.

Slide 4 - File Dialog Messages

There are only three messages sent from the file dialog:

FDM_ERROR - Sent when thhe dialog is about to display an error window. Trap this to display a custom error window.

FDM_FILTER - Sent before a file is added to the list box, allowing it to be filtered out.

FDM_VALIDATE - Sent after the user clicks on the OK button or double-clicks on a file, allowing validation.

Slide 5 - Using Standard File Dialog

This code fragment shows the basic technique for using the standard file dialog. Note the use of the standard OS/2 'trick' of memset()ting the memory of a structure to initialise all the elements to zero, NULL, etc., then setting the 'countbytes' element. Note also the double check for a valid hwndFileDlg and a return code of DID_OK.

Slide 6 - Standard File Open/Save Dialog

What can I say? A picture is worth . . .

Slide 7 - FONTDLG Structure

This is the structure which underlies the standard font dialog.

The students will need to understand why at least one, and possibly two hps's must be passed in the structure - yet presentation spaces have not yet been covered (P21D5S1 will cover). Basically a presentation space is a scratchpad where graphics information is combined to build up a picture. Fonts exist only within a presentation space.

A presentation space can be associated with a screen device context to paint in a window, or with a printer device context to paint on a printer. The two may (will) support different fonts: for example, the system supports Courier, Helvetica and Times (plus Symbol), while a printer may support Elite or its own unique fonts. If you query the fonts in a screen ps, you will get back system fonts, while querying the printer PS will return the printer fonts.

The standard font dialog can do both, allowing the user to enable/disable the display of printer fonts with a checkbox.

Frankly, the details of the other elements are best left for private study, using the online help and the header files. However, one cautionary note: depending on the version of the toolkit you are using the on-line help may be wrong! If in doubt, check the definition of the FONTDLG structure in the file \toolkt21\c\os2h\pmstddlg.h. This applies to OS/2 programming generally - the docs are sometimes in error, but the header files rarely are - and they are quite interesting to read through. If in doubt, use the definitions in the header files - at least that way, you and the C compiler will be in agreement.

Slide 8 - Font Dialog Flags

These flags are pretty well self-explanatory.

Slide 9 - Font Dialog Messages

These are mostly sent to a subclassing window procedure to notify it of changes to the face-name, style, color, etc. The FNTM_FILTERLIST message is sent to an application before the dialog window adds a facename, style or color to the appropriate listbox, allowing the application to filter their contents.

Slide 10 - Font Dialog Standard Controls

These ID's are mostly used by subclassing window procedures.

Slide 11 - Using the Font Dialog

This sample code fragment shows the simplest way to use the font dialog. Notice that the application must obtain a screen (actually, window) presentation space using WinGetPS() or GpiCreatePS().

Note also the use of the MAKEFIXED macro to construct a FIXED value for the structure element fxPointSize. The MAKEFIXED macro and FIXED type will be dealt with in the section on elementary GPI programming.

Slide 12 - Using the Font Dialog (cont)

It's perhaps worth mentioning that the owner of the dialog has to be the client window so that it can be sent notification messages. However, the parent is often HWND_DESKTOP, rather than the application's client window. This is because if the dialog was a child of the client, it would be clipped by it, and since some dialogs are quite big, they would not be fully visible. Centering on HWND_DESKTOP is a good default scheme.

Slide 13 - INI File Interaction

Title slide.

Slide 14 - INI Files

Everyone who uses OS/2 has a healthy respect for the two files \OS2\OS2.INI and \OS2\OS2SYS.INI, and learns to back them up frequently, and not mess with them! As a general rule, applications should utilise these files as little as possible, a good rule being to use an application-defined entry in OS2.INI to point to your application's own private INI file, which can be opened using the PrfOpenProfile() API and then manipulated using the usual Prf functions.

The major reason for avoiding using the INI files is that the Workplace Shell stores the object settings for all abstract objects in OS2.INI (that's why it gets so big). If your students don't know it, this is probably a

good time to cover the use of a CALL= statement in CONFIG.SYS to invoke CMD.EXE or even a batch file to back up CONFIG.SYS, the IN I files and STARTUP.CMD, coupled with the use of Alt-F1 on reboot to restore them.

Slide 15 - Contents of INI Files

I usually introduce this by using the Windows WIN.INI and SYSTEM.INI files as examples, since most folks have seen them. If not, you can use e.exe to open the ones in Win-OS/2. The structure of the OS/2 INI files is similar except that they get so large that they could not possibly be kept as text files for performance reasons. Instead, they are binary files which are indexed internally. The use of an API rather than direct access insulates client programs from future format changes (the internal format has changed twice already, with almost nothing getting broken).

The PrfQueryProfileString() API is relatively straightforward: pass it NULLs for the app and key names and it will return a list of application names. Pass it an application name and NULL key name and it will return a list of key names. Pass it the application name and the key name and it will return a list of values.

Slide 16 - Other Useful Functions

Self-explanatory. Note that non-PM applications can use the profile API as long as they call WinInitialize first in order to get an hab.

Day 5

Session 1 : GPI

Before starting the slide presentation I usually do a short demonstration section to illustrate some graphics concepts.

Start Windows Paintbrush (or restore it if you have it already started but minimised to avoid questions about slow loading of Windows apps). I usually draw a black diagonal line and a couple of intersecting rectangles. I point out that what this program stores internally is a bitmap - i.e. this pixel is black, this pixel is black, this pixel is white and so on, while pointing a different parts of the diagonal line.

The program does not have any concept of 'line' internally, so you can't pick up one end of the line and stretch it or move it elsewhere. Likewise, there is no concept of rectangle. The program does not know that 'these four lines' (point to them) make up a rectangle, or that 'these two lines' (point to them) cross at this point.

Illustrate what happens when you try to move one of the rectangles. Use the rectangular selection tool to crop as tightly around one of the intersecting rectangles as possible and move it; look what happens - it takes part of the other rectangle with it, since the program simply moves a rectangular group of pixels.

Now switch to a drawing program such as Corel Draw or Freelance Graphics (I use Freelance). Draw a line. Show that the line can be selected and its end points have handles which can be grabbed in order to stretch and move the line. Explain that the program has an internal data structure for each line: the structure definition (class) has fields for x and y position (pick up the line and move it), cx and cy offset to the other end (grab an end handle and stretch the line) and thickness, pattern and colour of line (double-click to produce the editing dialog for these). Each such instantiated structure is an object.

Now draw two overlapping rectangles. Explain that the program has a class structure for rectangles, which has fields for x and y position, cx and cy size, thickness and colour of border and colour and pattern of fill. You can demonstrate these by moving the rectangles, resizing and double-clicking to edit.

Now demonstrate that the two rectangles are separate objects and can be moved independently (i.e. changing the x and y fields of the structures). This is an object-oriented drawing program, whereas Paintbrush was a bitmapped program which did not understand objects.

Next, what about the object-oriented concept of inheritance? Can you create new classes which inherit from parent classes? As an example, some graphics systems have a class called roundrect: a roundrect is derived from a rectangle, but adds a field which is the radiusing of the corners. In Freelance, it's actually the reverse, rectangles are simply objects of class roundrect which have zero radius.

However, it is possible to illustrate a related concept: create a rectangle with rounded corners, duplicate it (Ctrl-F3), then modify the copy to have grey fill and send it to the background. Fill the foreground

roundrect with solid white and slide the background roundrect up and across to form a shadow. Use marquee selection to select both objects and group them.

Demonstrate that this new composite object can be duplicated, resized etc. This is an example of hierarchical picture composition: building complex picture elements out of simpler ones.

Next, pull down a menu and ask the audience to agree that "it drops down in front of the graphics image, right?". When they agree, point out that they are looking at a perfectly flat screen, and nothing drops down in front of anything else: the menu obliterates part of the graphics image. However, PM is smart enough to move that part of the image and store it in memory while the menu is present, and then restore it once the menu is dismissed. This feature is referred to as automatic picture repair.

But if the system is short of memory, preserving that part of the graphics image will force a disk swap. In this case, the system will not save the bitmap, but rather will let it be destroyed, invalidating a rectangle and then send a WM_PAINT message to the application, which must then repaint that part of the image.

And therein lies the next problem: how does the programmer decide which graphics objects have been damaged by the invalidation and therefore must be repainted? Repainting the entire graphics image could be much too time-consuming. In fact, the system provides functions to identify which objects encroach on the invalid region, in order to speed repainting. This process is called correlation.

Another aspect of correlation is 'hit detection', or input-to-picture correlation. When the user wants to select a line, he must click on it. But a line can be just one pixel wide, and with a high-resolution display and a low-resolution mouse, clicking on the line may be next to impossible. We must have a rule that says, 'near enough is good enough', work out when the user clicks the mouse, which is the nearest object, and is it near enough to be considered a hit? You can demonstrate this in (e.g.) Freelance by 'sneaking up' on the line - it will be selected when the mouse is still some distance from it.

Having completed these demonstrations, you can now start the slide-show.

Slide 1 - OS/2 Graphics Programming Interface

Intro slide

Slide 2 - GPI Primitives Can Draw . . .

Demonstrates the different kinds of graphics primitives supported by the GPI, for text, lines and arcs, and bitmaps.

Slide 3 - GPI Concepts

The points on this slide really relate back to the earlier demonstration. The GPI combines an abstract graphics 'language' with a device interface, which helps to make it abstract by decoupling devices.

Stored picture elements are the objects referred to above. Likewise hierarchical picture construction, picture editing/replacement was the duplication, resizing, movement, etc. done during the demo, and correlation and picture repair were also demonstrated.

The Device Interface essentially is the device driver architecture. The first two points essentially refer to the layered architecture of the DI. For example, if your plotter can draw a dashed line, then the DI lets it; but if the plotter lacks this capability, then the driver will draw a series of short line segments to produce the same effect. The DI can also provide information, such as paper size or fonts supported.

Slide 4 - PM's Roots

This slide shows the design influences on the development of PM and the GPI. Note that IBM mainframe standards like GDDM and 3270 Graphics Control Program are strongly represented, in part because it makes it easy for IBM to front-end mainframe applications under OS/2 (e.g. by providing 3270 emulation) but also because these programs have stood the test of time and work well.

PHIGS is an important standard in the UNIX and open systems world. IBM has recently also licensed OpenGL, the 3-D graphics library used in Silicon Graphics workstations, and will build it into OS/2.

Slide 5 - GPI is Different From Earlier Standards

GPI was the first system which is able to support both raster (across and down) devices, such as dot matrix printers and CRT displays, as well as vector devices, such as plotters, which can drop a pen and then scoot off in any direction including up the page. Notice that Postscript and other page description languages are vector in nature.

It also supports sharing of the display by means of overlapping windows, and printers by means of a sophisticated spooler subsystem. If you don't believe printing under PM is sophisticated, how come it merits an entire volume of the OS/2 Redbooks all to itself? The spooler caused major problems for both IBM and Microsoft in the development of OS/2 1.2, and was one of the (febrile) excuses MS used to dump OS/2 in favour of Windows.

Slide 6 - GPI Concepts (cont)

Time to draw a diagram. Start at the right-hand edge of the whiteboard. From top to bottom, draw a screen, a laser printer and a plotter (hey - if I can draw these, anybody can). Next, draw three boxes to their left - these are the presentation drivers (i.e. the graphics drivers, not the physical port drivers).

Now draw a box around each driver/device pair. That is a Device Context.

Screen device contexts are pretty obvious. Memory device contexts are mostly used for manipulation of bitmaps (scaling, bit-blitting, etc) before loading into a presentation space. The contents of memory device contents are not portable.

However, a metafile is essentially a portable graphic. Think of it as a tape recording of the graphics commands which build up an image. They can be collected in memory as a metafile, then saved to disk. When the metafile is reloaded and replayed, you'll get back the same image - on the same or a different device.

The audience may think of PM as being just part of OS/2, that runs on PC's, but this is incorrect for several reasons. First of all, it is used in operating systems other than OS/2, such as Unisys CTOS operating system. Hewlett-Packard has done a port of PM to UNIX. In addition, QMS has produced a PM

metafile printer. Send it a metafile, and it runs PM internally, replays the metafile and - click, whirrr, out comes the graphics image.

Incidentally, IBM LAN Server transports print jobs around the network as metafiles, and the OS/2 Picture Viewer can open the resulting spool files and 'print preview' them.

The 'other device' device context is the most commonly encountered DC for printers and plotters. However, loading a full device context can be very expensive (140 KB or so), particularly if you don't necessarily intend to print through it. For example, when you start a word processor program and it creates a new, default, document, one of the things it needs to know is the page size of the printer. So it will query the printer device context: expensive!

Hence the 'information' device context, which is designed specifically for this application. It is small, fast, and designed to support querying of fonts, paper sizes and other printer metrics.

Back to the drawing. To the left of the screen device context, draw another box and label it 'Presentation Space', with a line feeding the screen DC. Then, to the left of it, draw another box and label it 'Application', with a line to the PS. The application uses the GPI primitives to draw into the presentation space, and this in turn causes output on the associated device context - draw a simple line graph on the screen to symbolize this.

It is possible to bypass the PS and access the DC directly using the DevEscape() function, to perform tasks such as page ejects, mark start and end of jobs for the spooler, etc. User- (i.e. programmer-) defined escapes are also possible.

Slide 7 - Opening a Device Context

Very simple - this slide shows the functions and the attributes possible

Slide 8 - Presentation Spaces

Now, the secret of PM & GPI WYSIWYG operation. Back to the diagram.

Drawing into the Presentation Space can cause the following combinations of events:

Output on the device context, nothing stored in the PS

No output, but graphic segments stored in the PS

Output on the DC, graphics retained in the PS

I wonder if it's possible to have nothing drawn and nothing retained? Reminds me of a joke about winking at pretty girls in the dark, but it's not PC these days.

Once the PS contains retained graphics, it can be disassociated from the current DC (erase through the line) and reassociated with a different DC (draw a line to the printer DC). Now a GpiDrawChain() call will output the graphic on the selected device (draw the graph on the printer device).

Likewise it could be associated with the plotter DC (redraw the lines and draw the graph on the plotter).

Back to this slide. Remind the students that a graphics segment is nothing to do with a 286/386 memory segment. Symbol sets are like fonts; they are transformable in the same way as fonts. Line-type sets include thick lines, thin lines, red lines, blue lines, etc.

Draw controls are switches which control certain (five) global settings such as erasure of the window before painting. The color palette maps the internal representation of colours (24-bit - up to 16 million colors) to the physically realisable colors of the hardware.

The viewing pipeline is the various representations of the graphics image - there will typically be several of these.

Presentation space types: the system maintains a pool of 16 presentation spaces open at all times. These are called cached micro presentation spaces, and a handle to one can be obtained with the WinBeginPaint() API. A cached micro PS handle must always be released before returning from message processing.

A micro presentation space provides a lot of functionality, and its handle can be kept between messages. More importantly, its various modal settings (fonts, colors, position, etc.) are preserved. However, a normal PS is able to retain graphics - the others cannot. (Another type of PS, the AVIO PS, will be discussed later).

Slide 9 - GPS Segment Concepts

The picture chain is a list of segments. If no segments have been created, then the system puts all the primitives issued into a 'default' segment.

Segments normally chain from one to the other, but a segment can be called from another. For example, a drawing of a house may make four calls to a segment which draws a window.

A dynamic segment is used for animation, e.g. redrawing a graphics object which is being dragged around the window.

Segments are essential for retained graphics - anything drawn outside a segment is not retained, regardless of the drawing mode.

The graphics primitives generate elements, which are calls to graphics routines in the GPI (most of which are actually implemented in the presentation driver.

Graphics orders break down into three types: one-byte orders, two-byte orders and n-byte orders, which start with a count of the bytes that follow.

Slide 10 - Presentation Space Types

Note that cached-micro presentation spaces are not persistent, so that changes made in response to one message have no effect - the PS is reset.

Micro presentation spaces, by contrast, are persistent. Usually, the PS is created during WM_CREATE processing and destroyed during WM_DESTROY.

The normal PS provides full functionality, including retained graphics, and is most appropriate (in view of the memory expense) for WYSIWYG applications.

The AVIO PS is supported in 16-bit applications only. Basically, the AVIO PS supports the Vio- functions in a window, and provides a further transition for applications being ported from DOS, via a straight Vio port. This allows the application to now use a PM menu, and provides a text window which is black text on a white background.

Slide 11 - Presentation Page Units

Working in pixels is quite natural for some applications, while others need to have precise physical measurements.

Slide 12 - GPI Function Groups

Transforms: I spend some time on this topic, although it is dealt with in detail in the Advanced course. Done on the whiteboard:

Ask the audience this:

What's

```
            (1  0  0)
(1 2 0) x   (0  1  0)
            (0  0  1)
?
```

(There's always someone younger than you in the audience - they ought to remember this from school-days!)

The answer, of course is

```
( 1 )
( 2 )
( 0 )
```

Now, change the transform matrix to:

```
            (2  0  0)
(1 2 0) x   (0  2  0)
            (0  0  1)
```

Now, what's the answer?

(2)
(4)
(0)

This is an example of scaling - you've moved the point (1,2) to the point (2,4) - twices as far from the origin.

OK, now change it to:

$$(1\ 2\ 0) \times \begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Now what's the answer?

(-1)
(2)
(0)

In other words, the point has been reflected through the y-axis. Doing this to every point in a figure obviously reflects the entire figure.

Similarly, the matrix

$$(1\ 2\ 0) \times \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}$$

becomes

(2)
(3)
(0)

In other words, it has been translated 1 unit to the right and 1 unit up.

The most complex example is rotation:

$$(1\ 2\ 0) \times \begin{pmatrix} \cos(90) & \sin(90) & 0 \\ -\sin(90) & \cos(90) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

becomes

(-2)
(1)
(0)

In other words, the figure has been rotated through 90 degrees.

In order to be able to reduce scales, and support rotation, the matrix elements cannot be integers: they must support a fractional part. In the OS/2 GPI, the matrix is specified as a MATRIXLF structure (matrix with long and fixed components):

```
typedef struct _MATRIXLF {
FIXED    fxM11;  /* First element of first row  */
FIXED    fxM12;  /* Second element of first row  */
LONG     lM13;   /* Third element of first row  */
FIXED    fxM21;  /* First element of second row  */
FIXED    fxM22;  /* Second element of second row  */
LONG     lM23;   /* Third element of second row  */
LONG     lM31;   /* First element of third row  */
LONG     lM32;   /* Second element of third row  */
LONG     lM33;   /* Third element of third row  */
} MATRIXLF;
```

The FIXED type is 32 bits in size, and consists of a 16-bit two's- complement signed integer and an unsigned 16-bit fractional part. A FIXED can be constructed with the MAKEFIXED macro. For example, to obtain a value of 1.5 use:

```
MAKEFIXED(1,32768)
```

or alternatively

```
(LONG) 1.5 * 65536
```

Slide 13 - GPI Drawing Primitives

This slide is basically a catalog of useful functions. Point out that what each primitive does is either to produce the required graphic in the device context, store a graphics order in the presentation space, or both.

Slide 14 - Attributes on Primitives

The GPI is basically a modal graphics system. The programmer can set (for example) the current line width, colour, line end and so on, and these remain set for all drawing until next changed.

Attributes can be set independently for various sets of primitives: areas, lines, characters, images and markers. Although the attributes can be set individually using a variety of API's, the basic technique which will always work, and will allow setting of multiple attributes concurrently, is the GpiSetAttrs API. This takes as one of its arguments the address of a structure called a bundle which 'contains' all the attributes relating to one of the above graphic types. Two other parameters are bitfields which specify which parameters are to be set from the bundle and which are to be reset to the defaults. Point out that the operation of this function is conceptually similar to WinSetWindowPos - remember the SWP_ flags?

Slide 15 - Altering Single Attributes

This is another 'catalog' slide which lists APIs for setting individual attributes.

Slide 16 - Altering Common Attributes

Some attributes are common to all graphics types and do not apply specifically to lines, areas, etc. They can be altered using these functions.

Slide 17 - Coordinate Systems and Spaces

This slide illustrates the way in which an image is constructed from the various world space components into model space (optional), then into the presentation space and finally into the device space. The slide shows the various different transforms and API's between the different spaces

Slide 18 - Coordinate Systems and Spaces (cont)

This slide shows the clipping and scaling possible between the different spaces.

