Migration from OS/2

Presented by: **John Hall**

John Hall joined MS 3.5 years ago after recieving his Masters in Computer Science. His first major project was 'Windows Libraries for OS/2'. Currently, he is the leading OS/2 Migration expert at the MS Porting Lab. John has over 8 years experience in the industry.

Initial Investigations on the Subject of Porting from OS/2

There are several roads to Windows

Windows is a scaleable operating system, and currently exists in both 16bit and 32bit incarnations. In addition, there are usually several choices available for moving a given application to a particular Windows implemenation. Briefly, the options are:

- A Windows 3.x implemenation (16 bit)
- A 32 bit implementation on a Windows 3.x base (Win32S)
- Running an OS/2 character mode application without modification on Windows NT's OS/2 Subsystem.
- A Windows 32 implementation on Windows NT.

The focus of this presentation will be on the last option, though using the OS/2 Subsystem will be breifly touched upon.

Don't Panic

The original target of NT was to host POSIX and OS/2. In fact, many of the internal documents on the design of NT still refer to 'NT OS/2'. *Therefore* the functionality available in Windows NT is necessarily a superset of the functionality available in OS/2.

As far as PM, it is helpful to remember that the primary designers of the Windows user interface, the PM user interface, and now the Windows 32 user interface are all the same people. There are more similarities than differences between the two systems.

So *Don't Panic*. The situation is not as grave as it first appears.

Make use of the OS/2 Subsystem if Possible

Many corporate applications built on OS/2 are character mode applications and Windows NT includes an OS/2 1.x subsystem which supports them. If your application meets this description, then you do not have to port at all. (The OS/2 version of Brief is one of the most common programming editors used on Windows NT.) At worst, your character mode application may depend upon private device drivers. In this case, you would need to port your private device drivers to the Windows NT device driver model. It is then possible to communicate with these device drivers from your program running in the OS/2 subsystem.

Collect the Tools you Need

Whether you are already committed to a port to Windows NT, or you are seriously considering it, your first step should be to gather the tools necessary for the job. This should include:

- Windows NT SDK
- The OS/2 to Windows Migration Kit (098-35176) plus tool updates.
- A Compuserve account, to monitor and use MSWIN32:Porting from OS/2

Install the Windows NT SDK and build a few of the examples, explore the Windows 32 help file, then read the OS/2 to Windows Migration Kit overviews and build the tools provided. These two steps will help more than any lecture possibly could. Finally, the Compuserve forum is monitored by several people at Microsoft, including John Hall. Above all, don't be afraid to ask questions.

Know your Application -- and a tool explanation

Not all API's are created equal, some are more difficult to port than others, and sometimes the distribution and type of API's you have used in OS/2 is an important piece of information. Microsoft provides utilities to help obtain this information, analyze it, and prepare an initial porting estimate.

In the Migration toolkit you will find the 'apicount' utilities, a spreadsheet 'estimate.xls' and a macro file 'estimate.xlm'. Using the instructions for the apicount utility, run this on all of your OS/2 1.x .exe and .dll files. The final goal is to sum these values up and 'paste special -- values' these sums into the estimate.xls spreadsheet. When you are finished, the spreadsheet will look something like this:

Module Name			Exmpl	Time	Used	A-Api	A-Used	A-Time
Type		Totals:	5885	112	161	4171	70	44
OS/2 API Name	**Difficulty**	**Tran Rate**						
DevQueryCaps	**Major**	**1**	**2**	**2**	**1**	**0**	**0**	**0**
DosAllocSeg	**Minor**	**48**	**65**	**1.35**	**1**	**65**	**1**	**1.35**

In this example, the 'raw estimate' is approximately 112 man days to port an application containing 5,885 calls to 161 individual API's. Of these API's, 70 of them can be automatically converted to the Windows API set. These 70 API's are called 4,171 times by the propram for an estimated time savings of

44 man days.

Every public API in OS/2 is included in the spreadsheet, and some private ones are included as well. These API's are ranked by a 'difficulty' factor, which is a SWAG estimate of the difficulty in porting that API. For each difficulty category, an API/man day porting rate is given which results in the times listed. These rates are easily changeable in the spreadsheet if you feel the given rates are too generous.

Finally, an effort has been made to provide some measure of automatic translation of the most common and simpliest API's. If an API has an automatic translation, it is noted in the spreadsheet. Further discussion of the automatic translation is reserved for later.

Once you have looked at the basic information, try sorting the data with the 'Time' macro in estimate.xlm. This will sort the most time consuming API's in each category to the top. In a typical case, a disproportionate amount of the estimated time is located in just a few API calls. Very often, after looking at exactly how your application uses these API's you will decide the spreadsheet estimates are overstating the problem. Regardless, you will have a good idea where your problems might lie, and you can expend the most research time in those areas.

For the edification of the curious, in Microsoft's experience of collecting this information for a wide range of corporate customers, the following API distribution is common:

µ §

Microsoft's efforts to help port OS/2 applications have followed this distribution pattern. The research and documentation on how to port an OS/2 application is richest on the Dos*() and Win*() areas.

Starting the Port

A 12 Point program for OS/2 independence

1. Read all the steps. If you are using a multi-person project team, a number of the steps (after 6) can be done in parallel with little interaction.
2. Every member of your team will:
    - Install the Windows NT SDK
    - Be able to build the sample code and conversion utilities
    - Have access to Compuserve so they may ask questions.,
    3. The OS/2 source code should be assembled on one machine if at all possible, and built on that OS/2 machine. [Discovering you are missing pieces halfway through the process is

annoying.]

4. Modify all All make files to look *exactly like* those found in the conversion utilities or NT sample programs.
5. Run every source and header file through the automatic conversion utilities.
6. Manually change all MsgProc()s to **CALLBACK** MsgProc(HWND, **UINT**, DWORD, DWORD). (UINT and CALLBACK are changes).
7. Manually change the returns on all dialog boxes to the windows syntax (return TRUE or FALSE instead of using WinDefDlgProc()).
8. Modify all *.def files to look like the shrmem.def file in the conversion utilities.
9. Use the 'resconv' utility *and whatever manual hacking is required* to migrate all *.rc files from their OS/2 forms to Windows.
10. Provide each and every DLL with a 'LibMain' routine, even if it doesn't do anything. 'LibMain' routines are a vital part of Windows 32 and should be well understood.
11. If you have a large system that uses DLL's [more than 2 DLL's which reference functions in each other], it is *strongly advised* that you create a 'notimp.c' file for every exported function each DLL. The 'notimp.c' file should contain a fully parameterized dummy function for every exported function in the DLL [put up a message box saying "Not Implemented Yet!" and return an error].
12. Bring up a minimally functional main window as soon as possible, then gradually start adding functionality back in.

Notes on Step 4: Makefiles

One of the most common source of build problems are incorrect make files. Properly constructed Windows NT files are complex and very different from the make files you are accustomed to. Experience in the Porting lab has shown that the best method of creating a Windows NT make file is:

- Print a copy of your current make file.
- Delete your old make file.
- Now take a pre-existing sample make file (either exe or dll depending upon what you want to build) and edit that file to make it do what you want.

Notes on Step 5: Automatic conversion

There are two automatic conversion utilities: a sed script *script.sed* and the program *ms_ssed*. The procedure for converting a source code file (*.c) is:

- ren foo.c foo.os2 // save old OS/2 code
- sed -f script.sed < foo.os2 > foo.sed
- ms_ssed foo.sed foo.c // not needed for .h files.
- del foo.sed

script.sed is a sed script which preforms over 500 text replacements on OS/2 code to aid the conversion process. Although several API replacements are done in phase, experience has shown that modifying OS/2 manifest constants to their Windows equivalents have been the most important translations. Some of these are direct source code modifications, and others are done in *os2win.h* which is the include file that replaces *os2.h* in your source files.

ms_ssed, which is only run on source code files not header files, is a custom built macro translator which can handle more involved conversions than simple text replacement.

Finally, an OS/2 call is frequently replaced by a call to the *holeport dll*, which is designed to take OS/2 syntax for the function and attempt to do the right thing. You will probably wish to move away from the *holeport dll* at some point, but you can do it incrementally and use the code in the dll as an additional indication of how to deal with a particular API.

Notes on Step 9: .RC files

Resconv.exe is not a complete, robust solution to converting RC files. It chokes on 'unexpected' things which don't have equivalents in Windows. Usually, most of these involve OS/2 help and spin buttons -- but their are others. Just roll up your sleeves and get the job done.

Notes on Step 12: Bring up Main

I find it more pleasant to deal with an application which at least gets to the main window -- and much easier to debug as well. After performaing all of the global modifications suggested, just try and get the main window up. This may mean commenting out sections which don't compile (or introduce link errors) that you don't immediately know how to fix. After that, it your approach will probably be unique to your application.

Where are the Dragons?

Exit List Processing

The most 'dragon', dwarfing all other non-GUI issues, is Exit list processing. Exit list processing in OS/2 is most often installed by a DLL which allocates resources on behalf of a process and wishes to clean up upon process exit. This facility in OS/2 is limited, since the exit list processing only occurs in the context of thread 0 and a dll can not guarantee it will be executed. For example, a dynamically loaded dll can install an Exit list process, but if the dll is unloaded this process will not be executed. [It will be removed from the exit list.]

Windows 32 uses a different approach to exit list processing. Each dll has an entry/exit routine which is called every time a process or thread attaches to that dll and every time a process or thread detaches. This allows a dll to allocate and free resources on a thread level.

Early attention to where and why you were using Exit list processing in OS/2 will pay dividends.

Synchronization

Windows NT provides four major synchronization mechanisms: *Events* which replace the *DosSem* signaling capability, *Critical Sections* which perform like *Ram Semaphores, Mutex's* which are similar to the misnamed *FSR's* and **real** *Semaphores*. The overview section in the Migration kit should be read to familiarize yourself with the Windows 32 API's.

There is a dragon here -- OS/2 Critical Sections are not supported. The OS/2 Critical Section works by freezing all other threads in a process, something considered very rude in Windows NT. Algorithms depending upon this need

to be recoded.

InterProcess Communication

Windows 32 provides an exceptionally rich set of IPC mechanisms -- but it doesn't provide Signals. Anyone using signals for IPC will need to look at a different mechanism. Note that Signals are most often associated with trap handling, and Windows 32 uses a generic *exception handling* architecture for this purpose.

Shared Memory and Queues

Shared Memory under Windows NT is fundamentally different than shared memory under OS/2 -- and Windows NT doesn't provide Queues at all. In the conversion utilities you will find a shrmem.dll and a queue.dll, each complete with source, which provide these facilities in a manner similar to OS/2.

Thread ID's

In OS/2, the threads in an application had ID's starting at 0 for every process. In Windows NT, the thread id is unique across all processes. So if you are storing per-thread data in an array indexed by thread id it isn't going to work on Windows 32.

Dialog Boxes

It has been my obersvation that most corporate applications use dialog boxes very heavily -- API's such as *WinWindowFromID, WinSendDlgItemMsg, WinSetDlgItemText, WinDismissDlg, and WinQueryDlgItemText* form 5 out of the 8 most used calls in PM. (In total, those 8 API's account for 47% of all PM calls.) It is therefore fortunate that Dialog Boxes are relatively easy to migrate to Windows 32.

This largest change is the return sequence from a PM dialog box (using *WinDefDlgProc*) to the Windows dialog box sequence (TRUE or FALSE). If you have used controls other than the standard text, edit, push button types you might have some more work to do.

Frame Windows

A Windows Frame Window is not a collection of individual windows (controls) but one large window with special areas on it. The major difference here is you no longer have a *WM_CONTROL* message (they are always *WM_COMMAND*'s) and you no longer use something like *WinWindowFromID*() to get a control handle and manage it. In Windows, you (mostly) use the main window handle to manipulate the controls. Furthermore, the replacement for a PM message to a control is often a function in Windows.

Menus

Related to Frame Windows, the control and manipulation of Menus in Windows is different from PM. In general, in Windows an API is used to perform the operation that a *MM_* message is used for in PM. Currently, Holeport does its best to handle *MM_SETITEMATTR* (the most common menu message) and will look at supporting more of the menu messages in the future. However, it would pay to just rewrite these sections of your application.

A few other headaches

- Coordinate space origin is at the top left instead of bottom left.
- You no longer need manage HPS's. In general, a call that takes a PM DC or HPS will just take a DC in Windows.
- Message ordering is different. Expect code dependent upon this (particularly filtering code) to break. Note that with desynchronized message queues most of the reason for doing Peek's on particular message ranges disappears.

An Example Port

The Example Chosen

For illustration purposes, Petzold's head.exe program from *Programming the OS/2 Presentation Manager* by Microsoft Press was used. The full example can be found in the migration utilities.

Non-Automated Steps Required

In addition to the steps listed previously, this is a list of all of the other 'little' modifications which the port of head.exe required. More detail is available in the example than is reproduced here.

Modifications to the main routine:

- Windows programs start with WinMain() instead of main. The Holeport utilities include a WinMain which calls main(). Many windows functions require an instance handle, so WinMain sets a global hinst (if you roll your own WinMain, hinst must be global since some of the API translations depend upon it). It also sets a global nCmdShow which is the original Show command you should give your main window.
- After you create a window, you need to ShowWindow() and UpdateWindow().

- The main message loop is slightly different for Windows. First, you need *TranslateMessage*() to translate any accelerators. Second, you need to do a *PostQuitMessage*(0) when you get a *WM_DESTROY* message in your main Wndproc. This means you do not have to do a DestroyWindow on your main window, since DefWndProc does that in the *WM_QUIT* processing

The creation and management of Fonts is different for Windows and several modifications were necessary. To handle this, I did the following:

- Created *HP_SimpleFont*() which generates a font with a particular font name, size, and interprets the OS/2 FATTR *.font.attributes flags to do the right thing.
- The created HFONT is similar to the old LCID's used before, and that change required some fine tuning.

Finally, the syntax of Painting is a little different, and most of the time Windows does an auto-erase on the background for you. This required some minor modifications to the *WM_PAINT* processing.