Microsoft Corporation

Programming Considerations
For 32-Bit Windows Operating Systems

August, 1991

Abstract

The first prerelease Microsoft® Windows™ 32-Bit Software Development Kit will be available soon. During the beta testing of Windows 3.1, developers can update application source code and make changes that will result in a robust Windows 3.1 application as well as prepare the application for transition into the the full 32-bit environment provided by the Windows 32-Bit API. This paper is not a call to start creating 32-bit source code but rather a highlight of the changes that will benefit updating source code for Windows 3.1 now and Windows 32-Bit applications later.

1. Goals of Microsoft® Windows™ 32-Bit API

The creation of the Windows 32-Bit application programming interface (Windows 32 API) focused on five goals:

1. Provide a 32-bit migration path for existing Windows applications
2. Make porting a Windows application to Windows 32 as easy as possible
3. Create an efficient mapping layer to run Windows 3.x binaries on Windows 32-Bit systems
4. Support a single source code base for creating Windows 3.x and Windows 32-Bit binaries
5. Offer an identical Windows 32-Bit API on both Windows NT-mode and a future release of Windows Enhanced-mode

To achieve these goals, Microsoft derived the Windows 32-Bit API from the existing Windows 3.1 API, disallowing arbitrary name changes of data types, functions, and structures. At first glance, a Windows 32-Bit application is indistinguishable from an existing Windows 3.0 or 3.1 (hereafter referred to as Windows 3.x) application, both from an end-user's perspective and from a quick inspection of the source code. A native Windows 32-Bit application (unlike its cousin which uses the Windows 3.x API) can take full advantage of large linear memory allocation, multiple threads for background tasks and calculations, local and remote interprocess communication via named pipes, and other features detailed in ***The Windows 32-Bit API: An Overview.***

The Windows 32-Bit API will first appear on Windows NT for uniprocessor and multiprocessor Intel386™ and Intel486™ systems and for new RISC-based systems. A future version of Windows Enhanced-mode will also support the Windows 32-Bit API. All Windows 32 features are supported by both Windows NT-Mode and a future release of Windows Enhanced-mode, including linear address space, threads, and preemptive multitasking. Windows 32-Bit applications running Windows Enhanced-mode or Windows NT-mode will be binary compatible with Intel® processors and source compatible with Windows NT running on RISC processors.

This paper concentrates on two aspects of Windows 32:

1. Steps that developers can taken today while working on Windows 3.1 applications to better support binary compatibility of these applications on Windows NT;

2. Techniques that developers can use to create Windows code that is more portable and that will make it easier to create Windows 32-Bit versions of the application when Windows 32-Bit Software Development Kits are available.

2. Binary Compatibility

Windows 32-Bit systems will be able to run existing Windows 3.x applications with interoperability via dynamic data exchange (DDE), object linking and embedding (OLE), metafiles, and the clipboard with other Windows 3.x applications and with native Windows 32-Bit applications. Windows 3.x applications and Windows 32-Bit applications will exist side-by-side on the same display rather than running in separate screen groups. Windows 3.x applications will be fully compatible with Windows NT if developers follow these rules:

- Ensure that Windows 3.x applications run in Standard/Enhanced mode.
- Use published Windows 3.x APIs, messages, and structures.
- Do not modify WIN.INI directly; use a profile string API.
- Restrict direct port I/O (only standard devices).
- Do not directly access the disk controller; doing so is a security violation.
- Do not modify the system date and time; doing so is a security violation.

Windows NT provides U.S. government-level security capabilities that preclude allowing applications direct access to hardware. Rules 4-6 above do not apply to future versions of Windows Enhanced-mode that will support the Windows 32-Bit API. The restriction is a feature specific to Windows NT.

2.1 Design Requirements

Mapping-layer technology has been offered in the past to allow Windows-based applications to run on OS/2®. Past solutions such as Windows Libraries for OS/2 (WLO) required special runtime libraries and DLLs before Windows-based applications could run on OS/2. ISVs must ship WLO mapping-layer DLLs along with their applications, which complicates distributing and installing them. This approach is unacceptable on Windows 32-Bit systems.

To achieve binary compatibility and high performance on Windows 32-Bit systems, developers of Windows version 3.x applications:

- do not need to recompile the source code,
- do not need to use special runtime libraries,
- and do not need to develop or acquire special tools to make executables compatible.

The ability to run Windows 3.x binaries lets a developer update to Windows 32-Bit systems and continue to use existing Windows 3.x applications as well as native Windows 32-Bit applications as they become available. Native Windows 32-Bit applications will take advantage of the higher performance linear 32-bit addressing and enormous capacity increase for data processing. Microsoft will encourage Windows developers to test their products on pre-release versions of Windows NT through a Windows NT beta test program.

2.2 Supported Features

The following is a list of many of Windows 3.x features inherently supported on Windows 32-Bit systems. This list shows that existing Windows-applications can be binary compatible with future Windows 32-Bit systems with little work on the part of developers. It also illustrates that complex windowing, graphics, and low-level operating system reliance by Windows 3.x binaries will be completely supported.

Major user interface features that are fully supported include:

- Multiple document interface messages and default message handling
- Resource files (e.g. dialog boxes, menus, accelerator tables, and user-defined resources)
- DDE messages and the DDE manager library (DDEML) API
- Windows version 3.1 OLE
- Metafiles
- Clipboard data exchange

Major graphical interface features that are fully supported include:

- TrueType and TrueType APIs
- Windows 3.x icons and cursors in existing format
- Bitmaps (BMPs) and device independent bitmaps (DIBs)
- Printing by means of native Windows 32-Bit printer drivers

Base system functionality includes support for:

- Shared memory for interprocess communication
- NetBIOS and Microsoft LAN Manager for DOS named pipe support
- MS-DOS 5.0 interfaces (called with **DOS3Call** or **INT21**)

2.3 Methods to Achieve Binary Compatibility

A Windows 3.x application must meet several requirements to be binary compatible on Windows NT; most relate to security. Applications do not "own" the machine as they do in Windows 3.x. As discussed above, directly accessing hardware (without a device driver) is a severe security violation on Windows NT. If direct hardware manipulation were supported, aberrant or malicious applications could affect the hardware causing system crashes or system instablity. This possibility is not acceptable in mission critical applications or on network servers, which Windows NT is designed to support.

The Windows 32-Bit system will employ a registration database which will maintain all system and application configuration information. Files such as WIN.INI will no longer exist in the file system; instead, calls to the profile API (for example, **GetProfileString**) will be routed to the database. Therefore, applications should not attempt to create or modify *.INI files directly by means of file I/O API. The Windows 3.x profile APIs should manipulate all profile information. Installation programs that create private installation files should be modified to use the profile API.

Applications must be compatible with Windows 3.x Standard- or Enhanced-mode. Windows 32-Bit systems will not support Windows real mode. Applications should use only published Windows 3.x APIs, messages, and structures.

3. Portable Coding Techniques

With the release of Windows 3.1, many applications are being updated to add support for features such as OLE and to take advantage of TrueType. Because Windows programmers are already scrutinizing their applications' sources, now is a convenient time to prepare the code for the future, which offers a 32-bit environment with powerful new features.

The discussion below concentrates on the important issues that affect the portability of existing Windows source code to Windows 32-Bit. Although this list may seem long and detailed, all recommendations are useful for creating robust Windows 3.1 applications. In addition, applications will be more portable, and it will be easier to create native Windows 32-Bit applications quicker when Windows 32-Bit Software Development Kits are available.

Rules for Writing Portable Windows 3.x/Windows 32-Bit Source Code

- Parse *wParam* and *lParam* immediately in **WndProc** routines.
- NULL is a valid return value from **GetFocus** and **GetActiveWindow**.
- Use **FindWindow** instead of **hPrevInstance** to find other running instances.
- **GlobalLock** and **malloc** will not return 64K aligned pointers.
- Use Windows 3.x DIB functions to initialize color bitmaps.
- Do not use **GetInstanceData;** replace with supported IPC mechanism.
- Do not share GDI object handles (e.g. pens, bitmaps) between processes.
- Compile warning level -W2 or higher (-W3 recommended).
- Create function prototypes for all functions.
- Review structure member alignment and data types.
- Remove hardcoded buffer sizes (for example, file names and path names).
- Do not extract private copies of WINDOWS.H definitions.
- Use unique typdefs (HPEN, HWND, not HANDLE or int).

3.1 A Brief Look at Windows 32-Bit

If you start with Windows 3.x source code, creating a native Windows 32-Bit application using the Windows 32-Bit API is straightforward and requires minimal source changes. In general, the Windows 32-Bit API simply involves widening parameters and return values to 32 bits. Over the course of a few months, Microsoft ported a range of Windows 3.x source code to Windows 32-Bit, including the complete Windows 3.0 and Beta 3.1 software development kit sample code and relatively complex Windows 3.1 applets--Program Manager, File Manager, Cardfile, and so on. This porting effort has validated the design of the Windows 32-Bit API and proven that it is possible to quickly create Windows 32-Bit applications from Windows 3.x sources. As an additional exercise, Microsoft modified the Windows software development kit and system applet source code to be portable, allowing Windows 3.1 and Windows 32-Bit binaries to be created from the same code base.

The Windows 3.1 system applets contain more than 100,000 lines of source code. File Manager in particular contains approximately 20,000 lines of code; yet within one day it was compiling as a native Windows 32-Bit application. Within a week, the File Manager could execute and display directory listings. Changes included recoding several assembler routines in C so that the sources can be compiled for both Intel and RISC processors. Few changes to the original Windows 3.x C code were required, which is indicated by the short time needed to create a functional, portable version of the File Manager.

An important porting factor is that Windows 3.x resource files containing menus, dialog boxes, icons, accelerator tables, and so on are directly compatible with the 32-bit Resource Compiler. You need not modify the resource files for Windows 32-Bit. This is not surprising, since the resource file is simply a script with no information that is 32-Bit sensitive.

3.2 Windows 32-Bit Sample Source Code

The code fragment below is from the Windows 3.0 software development kit sample, GENERIC. This code fragment compiles without modification with either Windows 3.x or Windows 32-Bit development tools. Only one minor change to the entire GENERIC sample is required; the fragment builds completely as either a Windows 3.x or Windows 32-Bit binary. Although the GENERIC sample is not particularly sophisticated, it does contain a menu and a dialog box, indicating that complex Windows functionality is easily supported.

If we look at the code fragment through the eyes of Windows 32-Bit, we see a true 32-bit application. Function parameters, pointers, and structure members all widened from 16 to 32 bits. This widening is accomplished "under the covers" by means of typedefs in WINDOWS.H (Windows 32-Bit version). For example, the typedef LPSTR is a linear 32-bit pointer. The variable *hInst* is defined as a 32-bit HANDLE. The window handle, *hWnd*, returned by **CreateWindow,** is a 32-bit window handle. The window class structure contains 32-bit handles to icons, 32-bit linear pointers to string constants, and a 32-bit stock brush handle.

Generic Sample Application form Windows 3.0 SDK

```
#include "windows.h"                    /* required for all Windows applications*/
#include "generic.h"                    /* specific to this program      */

HANDLE hInst;                           /* current instance      */

int PASCAL WinMain(hInstance, hPrevInstance, lpCmdLine, nCmdShow)
HANDLE hInstance;                       /* current instance       */
HANDLE hPrevInstance;                   /* previous instance      */
LPSTR lpCmdLine;                        /* command line      */
int nCmdShow;
                                            /* show-window type (open/icon) */
{
     MSG msg;
                                            /* message       */

     if (!hPrevInstance)                /* Other instances of app running? */
if (!InitApplication(hInstance))        /* Initialize shared things */
     return (FALSE);                    /* Exits if unable to initialize       */

     /* Perform initializations that apply to a specific instance */

     if (!InitInstance(hInstance, nCmdShow))
          return (FALSE);

     /* Acquire and dispatch messages until a WM_QUIT message is received. */

     while (GetMessage(&msg,            /* message structure      */
     NULL,                              /* handle of window receivieng thssage */
     NULL,                              /* lowest message to examine       */
     NULL))                             /* highest message to examine       */
{
TranslateMessage(&msg);                 /* Translates virtual key codes      */
DispatchMessage(&msg);                  /* Dispatches message to window      */
     }
     return (msg.wParam);               /* Returns the value from PostQuitMessage */
}

BOOL InitApplication(hInstance)
HANDLE hInstance;                       /* current instance       */
{
     WNDCLASS   wc;
```

```
 /* Fill in window class structure with parameters that describe the          */
      /* main window.          */

      wc.style = NULL;                                                        /* Class style(s).      */
      wc.lpfnWndProc = MainWndProc;          /* Function to retrieve messages for   */
                                             /* windows of this class.              */
      wc.cbClsExtra = 0;                                                      /* No per-class extra
data.               */
      wc.cbWndExtra = 0;                                                      /* No per-window extra
data.              */
      wc.hInstance = hInstance;                                               /* Application that owns
the class.    */
      wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
      wc.hCursor = LoadCursor(NULL, IDC_ARROW);
      wc.hbrBackground = GetStockObject(WHITE_BRUSH);
      wc.lpszMenuName =   "GenericMenu";       /* Name of menu resource in .RC file. */
      wc.lpszClassName = "GenericWClass";   /* Name used in call to CreateWindow. */

      /* Register the window class and return success/failure code. */
      return (RegisterClass(&wc));
}

BOOL InitInstance(hInstance, nCmdShow)
      HANDLE               hInstance;                                         /* Current instance
identifier.          */
      int                  nCmdShow;                                          /* Param for first
ShowWindow() call. */
{
      HWND                 hWnd;                                              /* Main window handle.
*/

      /* Save the instance handle in static variable, which will be used in   */
      /* many subsequence calls from this application to Windows.                */

      hInst = hInstance;

      /* Create a main window for this application instance.   */
      hWnd = CreateWindow(
            "GenericWClass",                                                  /* See RegisterClass()
call.               */
            "Generic Sample Application",                                     /* Text for window title
bar.              */
            WS_OVERLAPPEDWINDOW,          /* Window style.                      */
            CW_USEDEFAULT,                                                    /* Default hoorizontal
position.          */
            CW_USEDEFAULT,                                                    /* Default vertical
position.             */
            CW_USEDEFAULT,                                                    /* Default width.
*/
            CW_USEDEFAULT,                                                    /* Default height.
*/
            NULL,                                                             /* Overlapped windows
have no parent. */
```

```
        NULL,                                                   /* Use the window class
menu.           */
        hInstance,              /* This instance owns this window.      */
        NULL                                                    /* Pointer not needed.
*/
    );

    /* If window could not be created, return "failure" */
    if (!hWnd)
        return (FALSE);

    /* Make the window visible; update its client area; and return "success" */
    ShowWindow(hWnd, nCmdShow);                                 /* Show the window
*/
    UpdateWindow(hWnd);             /* Sends WM_PAINT message                   */
    return (TRUE);                  /* Returns the value from PostQuitMessage */
}
```

3.3 User Interface Code

3.3.1 Message Parameter Packing

With the widening of handles to 32 bits, both *wParam* and *lParam* (the additional message parameters) must be 32 bits wide. The impact is to *wParam* because *lParam* is already 32 bits wide in Windows 3.x. If in Windows 3.x applications a handle and another value were packed into the high and low 16 bits of *lParam*, widening to 32 bits requires repacking some values. A 32-bit handle occupies *lParam* completely, requiring the previously packed second parameter to be moved to *wParam*. A few messages have been affected by handle widening, including WM_COMMAND, which is illustrated below:

WM_COMMAND

Win 3.x:	wParam == window id lParam == hwnd, command
Win 32-Bit:	wParam == window id, command lParam == hwnd

The WM_COMMAND *window id* and *command* parameters remain 16 bit values in Windows 32-Bit and can therefore be packed in the widened 32-bit *wParam*. The 32-bit *hwnd* value is now fully contained in *lParam*. To minimize the affect of parameter packing differences, a set of macros has been created that parse message parameters. In this way, you can compile source code either as a Windows 3.x application or as a Windows 32-Bit application without unique message handling code or C compiler `#ifdef` directives. These macros will be in the Windows 32-Bit software development kit. Examples of macros used to parse WM_COMMAND information are:

```
GET_WM_COMMAND_ID   (wParam, lParam) // Parse control ID value
```

```
GET_WM_COMMAND_HWND(wParam, lParam) // Parse control HWND
GET_WM_COMMAND_CMD (wParam, lParam) // Parse notification command
```

The underlying macro definitions are Windows 3.x and Windows 32-Bit specific; parsing the information from *wParam* or *lParam* as appropriate for each implementation. The imporant point is that you can easily create readable source code that can be compiled for Windows 3.x or Windows 32-Bit.

3.3.2 Parsing wParam/lParam

Because of the parameter repacking of several Windows messages, parse (extract) values from *wParam* and *lParam* in the message handling window procedure rather than passing parameters to worker functions where extraction occurs. If only *wParam* or *lParam* is passed to the worker routine, under Windows 32-Bit either parameter may no longer contain the value required by the worker routine. Delayed extraction of message parameter data in worker routines is not always easy to spot. Extracting the value within the message handling procedure localizes the impact of Windows 32-Bit message repacking.

An alternative solution is to always pass both *wParam* and *lParam* to worker routines. In this way, the routine will always have access to parameter values, whether compiled for Windows 3.x or Windows 32-Bit. This solution is less efficient because unused data is being transferred on the stack.

3.3.3 Profile String Usage

Systems offering Windows 32-Bit support will also provide a registration database. All system and application configuration data will be stored in the database on a per user basis with appropriate security controls to assure that applications cannot corrupt one another's data or the system's configuration data. A centralized database has a number of advantages, including simpler installation, remote administration of workstation software, remote software updating, and error logging.

Windows 32-Bit versions of the Windows 3.x profile API (for example, **GetProfileString**, **WriteProfileString**) route profile string requests, including private profiles (that is, *.INI files) to the registration database transparently. Therefore, do not attempt to manipulate *.INI files directly with file I/O functions. These files will not exist, and the data contained in them is not accessible via file I/O calls; only the profile string API will be supported.

3.3.4 Localized Input

The Windows 32-bit model is different from Windows 3.x in that input ownership is assigned at user input time -- when the input is created -- instead of when the input is read out of the system queue. For this reason, each thread has its own input-synchronized state information. In other words, each thread has its own input-synchronized picture of the mouse capture and the active window and is aware of which window has the focus.
This change adds tremendous benefit to programmers and users alike. It is no longer possible for an application that fails to process messages to bottleneck the system. Unlike Windows 3.x

and OS/2 Presentation Manager, no applications will be affected by other applications that process their messages slowly or who otherwise fail to check their message queue.

The following APIs are affected by localized input state:

```
SetFocus( HWND )
GetFocus( VOID )
SetActiveWindow( HWND )
GetActiveWindow( VOID )
GetCapture( VOID )
SetCapture( HWND )
ReleaseCapture( VOID )
```

In general, the **Get** APIs query only local current thread state. The **Set** APIs set state local to the window creator thread. If the current thread did not create the window, then the current thread's related input state is set to NULL as if the input related state were being transferred between threads.

Thus, the Windows 3.x semantics of APIs that return input-synchronized state are changed slightly. For example, **SetFocus** can be called with an *hwnd* and return TRUE for success, but a follow-up call to **GetFocus** might return NULL. More substantially, **GetFocus** now returns NULL if the calling thread does not have a focus window. Under Windows 3.x, **GetFocus** never returns NULL because a window in the system always has the keyboard focus.

Therefore, code applications to expect that functions such as **GetFocus** can return NULL as a legal value. The return value should be tested against NULL before being used in subsequent function calls.

Mouse capture is affected in an added dimension. The Windows 32-Bit server input thread cannot know ahead of time when an input thread will set the capture. Also, regardless of the input state of any application, the system must allow the user direct input to any other application at any time. Therefore, the semantics of mouse capture change slightly.

The semantics of how and when the capture changes are not affected; how and when an application gets mouse input is affected. The Windows 32-Bit server will send all mouse input between a mouse down operation and a mouse up operation to the queue of the thread that created the window into which the original mouse down went. Thus, the input thread processes mouse capture as the input is read out of the queue. If the mouse button is down during the mouse capture, the capture window sees all input generated by the mouse, no matter where the mouse is on the screen, until the mouse button goes up or the mouse capture is released. If a thread sets the mouse capture while the mouse button is up, the mouse capture window sees mouse events only as long as the mouse is over a window that thread created.

3.4 Graphics Interface Code

3.4.1 DIBs vs DDBs

Beginning with Windows 3.0, device independent bitmaps (DIBs) have been the recommended format for creating and initializing bitmap data. This format includes a header with bitmap dimensions, color resolution, and palette information supporting portability between Windows 3.x and Windows 32-Bit. Device dependent bitmaps (DDBs), as originally offered in Windows 1.x and 2.x, are not recommended.

Because DDBs lack complete header information, applications that directly manipulate DDB data are not portable to Windows 32-Bit. Developers are encouraged to write to the Windows 3.x DIB API (for example, **SetDIBitmapBits**); these calls are portable. Windows 32-Bit does provides, however, a subset of functionality for DDB API, such as **SetBitmapBits**:

- Monochrome DDBs are fully supported
- Caching color bitmaps with **GetBitmapBits** and **SetBitmapBits** is supported

Caching implies that **GetBitmapBits** is used to save bitmap data on disk. Under low memory situations in Windows versions 1.x and 2.x, the bitmap could be destroyed and easily restored with GetBitmapBits. This implies that the DDB data is never manipulated; it is simply backed up and restored on disk in its original form. While caching is not needed in Windows 32-Bit, source code employing this technique will still be supported.

Windows 32-Bit does not support initializing color DDBs with **CreateBitmap.** Such code is also not portable among Windows 3.x systems with different display drivers because DDB data is device driver dependent.

3.4.2 Sharing Graphical Objects

Windows 3.x runs all Windows applications in a shared address space. Data can be directly manipulated, and other Windows processes can directly access per-process objects that the system created. This architecture has been exploited by some applications that create a single graphical object, such as a pen or a bitmap, and allow separate processes to use the pen or draw on the bitmap.

Windows 32-Bit applications run in separate address spaces, and graphical objects are owned by the process that creates them. Only its owner can manipulate a graphical object. A handle to a bitmap passed to another process cannot be used by that process because the original process retains ownership of the bitmap.

Pens and brushes should be created by each process. A cooperative process may access the bitmap data in shared memory (via standard interprocess communications) and create its own

copy of the bitmap. Alterations to the bitmap must be communicated between the cooperative processes via interprocess communication and a proper protocol. One such protocol is DDE. Windows 32-Bit will add an explicit ownership transfer API for graphical objects to explicitly allow cooperative applications to share graphical objects.

3.5 Base System Support

3.5.1 Instance Initialization

The first release of Windows (version 1.01) was designed to run in 8088-based systems, which assumed limited installed memory (512K RAM). Functions such as **GetInstanceData** and knowledge about other instances of an application already running allowed efficient data sharing and initialization using data belonging to other running instances. On secure systems in which applications run in separate address spaces, these functions no longer are appropriate.

Therefore, applications that want to share data among several instances must replace calls to **GetInstanceData** with standard interprocess communication techniques such as shared memory and/or DDE.

A Windows 32-Bit version of **WinMain** supports the same parameter list as does Windows 3.x:

```
int WinMain(hInstance, hPrevInstance, lpCmdLine, nCmdShow)
```

However, the parameter *hPrevInstance* always returns NULL, indicating that this is the first instance of the application, regardless of any other already-running instances. Although this situation would appear to be a problem, the initialization of most applications is handled correctly. Under Windows 3.x, multiple instances can share private window classes registered by the first instance. Under Windows 32-Bit, each instance is required to register it own window classes.

Applications usually test *hPrevInstance* to see if they must register their window class. This test is guaranteed to work optimally under Windows 32-Bit, always indicating the first instance of the application, and Windows 32-Bit requires that every instance register its own window classes.

Some applications, however, must know if other instances are running. Sometimes data sharing is required, but typically applications that care about multiple instances are interested in assuring that only one instance of the application runs at any time. An example is the Control Panel; another is the Task Manager.

Applications such as these cannot use *hPrevInstance* in Windows 32-Bit to test for previous instances. These applications must use an alternative method, such as creating a unique named pipe, broadcasting a unique message, or calling **FindWindow.** If another instance is found, the application determines which instance should be terminated.

3.5.2 Memory Manipulation

Under the Windows 3.x segmented memory architecture, globally allocated memory always aligns on segment boundaries. Both **GlobalAlloc** and the C runtime **malloc** family of functions allocate global memory in a way that the 16-bit offset of the 32-bit segmented pointer that references the base address of this data is always 0.

This behavior is not portable to linear memory. Memory allocation is not guaranteed to align on 64K boundaries. Memory is allocated with a 4K page granularity, but some objects may be packed to fit within a single page to maximize memory efficiency. (See below for more information about pointer manipulation.)

3.5.3 Windows 32-Bit API Replacements for Int 21

Direct **INT21** calls or the use of the Windows 3.x **DOS3Call** API to request MS-DOS to perform file I/O operations must be replaced by the appropriate Windows 32-Bit file I/O calls. Windows 32-Bit offers a complete set of named APIs to replace nonportable **INT21** calls including:

INT 21H Function	DOS Operation	Windows 32-Bit API Equivalent
0EH	Select Disk	SetCurrentDirectory
19H	Get Current Disk	GetCurrentDirectory
2AH	Get Date	GetDateAndTime
2BH	Set Date	SetDateAndTime
2CH	Get Time	GetDateAndTime
2DH	Set Time	SetDateAndTime
36H	Get Disk Free Space	GetDiskFreeSpace
39H	Create Directory	CreateDirectory
3AH	Remove Directory	RemoveDirectory
3BH	Set Current Directory	SetCurrentDirectory
3CH	Create Handle	CreateFile
3DH	Open Handle	OpenFile
3EH	Close Handle	CloseHandle
3FH	Read Handle	ReadFile
40H	Write Handle	WriteFile
41H	Delete File	DeleteFile
42H	Move File Pointer	SetFilePointer
43H	Get File Attributes	GetAttributesFile
43H	Set File Attributes	SetAttributesFile
47H	Get Current Directory	GetCurrentDirectory
4EH	Find First File	FindFirstFile
4FH	Find Next File	FindNextFile
56H	Change Directory Entry	MoveFile
57H	Get Date/Time of File	GetDateAndTimeFile
57H	Set Date/Time of File	SetDataAndTimeFile
59H	Get Extended Error	GetLastError
5AH	Create Unique File	GetTempFileName
5BH	Create New File	CreateFile
5CH	Lock	LockFile
5CH	Unlock	UnlockFile
67H	Set Handle Count	SetHandleCount

In most situations, the standard C runtime libraries are sufficient for normal file I/O. The C runtime has the advantage of being portable across many platforms.

3.5.4 Dynamic Link Libraries

DLL initialization and termination functions behave differently in Windows 3.xand Windows 32-Bit in terms of how they are defined, when they are called, and the information that is made available to them. Windows 32-Bit DLLs are easier to create and have functionality not currently available in Windows 3.x. In Windows 3.x, initialization and termination functions must be provided: the termination function must be named WEP, and the initialization function is the DLL MASM entry point. Initialization and termination functions are optional in Windows 32-Bit DLLs.

In Windows 3.x, the DLL initialization function is called once, when the DLL is first loaded in the system. The function is not called again, even if other applications that use the DLL are invoked. Likewise, the DLL termination function is not called until the DLL is unloaded from the system, when the last application using it terminates or frees the library. The initialization and termination functions are distinct. The startup code for the initialization function must be in assembly language, to allow access to parameters that are passed in machine registers.

In Windows 32-Bit, the DLL initialization function is the same as the termination function, and its name is specified at link time. Initialization or termination functionality is selected by a Boolean parameter, *bAttaching*, passed to the initialization function. The DLL initialization function is called each time a process attaches to the DLL for the first time or detaches from the DLL for the last time. Thus, if five processes access the same DLL, the DLL's initialization function is invoked five times with the *bAttaching* parameter set to TRUE. When these five processes terminate, detaching the DLL from each process causes five calls to the DLL initialization function, with the *bAttaching* parameter set to FALSE.

Windows 3.x DLLs are typically implemented completely in assembly language or in C and linked to the standard LIBENTRY.ASM function. This function calls **LibMain** after initializing the heap and saving appropriate registers. In porting to Windows 32-Bit, DLLs implemented in assembly language should be rewritten in C so that they are portable to RISC-based systems.

Windows 3.x DLL initialization functions are passed the following information:

- the DLL's instance handle
- the DLL's data segment (DS)
- the heap size specified in the DLL's .DEF file
- the command line

Windows 32-Bit DLL initialization functions are passed the following information:

- the DLL's module handle
- the *bAttaching* Boolean, selecting initialization or termination

The Windows 32-Bit module handle is analogous to the Windows 3.x instance handle. In Windows 32-Bit, the data segment is irrelevant because declared DLL data is either private to each process accessing the DLL or shared among cooperative processes accessing the DLL. The DLL's module definition file controls whether DLL data is shared or private. The heap size is not passed to the Windows 32-Bit DLL initialization function because all calls to local memory management functions operate on the default heap, which is provided to each process. The command line does not need to be passed as a parameter since it can be obtained under Windows 32-Bit through an API function.

Although the Windows 3.x LIBENTRY.ASM routine contains nonportable assembly routines, it isolates the assembly language initialization and supports writing additional DLL-specific initialization in C via the LibMain routine. For portability to Windows 32-Bit, DLL initialization code should be added to the **LibMain** routine and written in C. (For further information on LIBENTRY.ASM, see the Windows 3.x software development kit documentation.).

3.6 C Coding Guidelines

The Windows 32-Bit API was designed to simplify the creation of Windows 32-Bit applications from Windows 3.x sources. Specific API differences have been discussed above. Creating portable Windows code also involves writing portable C. Fortunately, the similarity of Windows 3.x and Windows 32-Bit requires only that a concise set of portable C guidelines be followed. Windows programs have generally been optimized to operate with the segmented Intel architecture. Therefore, the change from segmented to linear memory is the most significant issue in creating portable C code

3.6.1 Pointer Manipulation

Windows 32-Bit supports a compatible set of memory management functions, such as **GlobalAlloc** and **GlobalLock**, and a new set of advanced linear memory APIs. Therefore, existing Windows applications can easily be converted to Windows 32-Bit and continue to use the Windows 3.x memory allocation and handle dereferencing API.

As mentioned previously, memory allocations are not aligned on 64K boundaries. Therefore, any pointer arithmetic based on assumptions of segment:offset encoded pointers will fail in Windows 32-Bit. When computing offsets to arrays of structures, do not create pointers by combining a computed 16-bit offset with the high-order 16 bits of an address pointer. This type of pointer arithmetic depends on segment:offset encoded addresses.

Several other pointer characteristics should be observed:

- All pointers (even pointers to objects in the local heap) grow to 32 bits
- Code that takes advantage of 16-bit pointer address wrapping is not appropriate with linear addresses
- Structures that hold NEAR pointers in Windows 3.x will grow from 2 bytes to 4 bytes in Windows 32-Bit

3.6.2 Promotions and Ranges

Expressions involving the C integer data types (int and unsigned int) should be reviewed for portability, especially if the compiler already generates warnings about signed/unsigned mismatches or conversion warnings. The int data-type grows from 16 to 32 bits, which can subtly affect applications compiled for Windows 32-Bit. Typical problems encountered are sign extensions and assumptions (sometimes unintentional) about ranges. Loops that take advantage of 16-bit ints and of the fact that integer loop counters will wrap at 32767 or 65535 will experience problems when the integer loop counters grow to 32-bit and wrapping occurs at 2GB or 4GB.

3.6.3 Structure Member Alignment

Data accesses to unnaturally aligned data elements are expensive on some hardware architectures and are illegal on others. For example, on the Intel 80386 accessing a DWORD that is not 4-byte aligned results in a performance penalty. When the same code is moved to a MIPS RISC processor, the misaligned access generates a fault. The system handles the fault, and system software decodes the data. Although the code is portable, it is not efficient. Therefore, all data elements should be aligned consistently with their type. Alignment rules vary with architecture, but the following guidelines are appropriate for the Intel and MIPS processors targeted by Windows 32-Bit:

Windows 32-Bit Structure Member Alignment

char:	Align on byte boundaries
short (16-bit):	Align on even byte boundaries
int/long (32-bit):	Align on 32-bit boundaries
float/double:	Align on 32-bit boundaries
structures:	Align on 32-bit boundaries

Creating a portable structure that is both efficient in memory usage (without packing) and aligned properly is possible.

3.6.4 Unique Typedefs

As illustrated in the GENERIC code fragment listed earlier in this document, unique typedefs are useful in creating portable code. Even though the typedefs can have different underlying definitions in Windows 3.x and Windows 32-bit, Windows source code can remain unchanged.

Windows offers unique typedefs for most objects defined in WINDOWS.H. Unique typedefs such as HPEN, HBRUSH, and HWND better support portability to Windows 32-Bit than generic typedefs such as HANDLE. Although all handles in Windows 3.x are interchangeable with HANDLE or unsigned int, using these basic data types affects porting to Windows 32-Bit because various objects require different typedefs under Windows 32-Bit than under Windows 3.x.

Just as using unique typedefs is recommended when defining (or casting) Windows objects, creating a complete set of unique typedefs for application-specific objects is also strongly recommended. As with the Windows objects, the underlying application-specific data types and structures can be modified and minimally affect source code that uses these data types.

3.6.5 General Recommendations

The following coding recommendations are well known but are occasionally ignored. Reviewing your code and addressing the following issues will create more robust Windows 3.1 code and will create code that is more easily ported to Windows 32-Bit.

Review hard-coded buffer sizes for file names and environment strings. Although dynamically allocating buffers to hold strings is not necessary, Windows 32-Bit supports long file names (256 characters). Therefore, buffers hard coded assuming FAT 8.3 format will not take advantage of long file name support. Using a #define to define sizes for array allocations will assist portability of the source code to Windows 32-Bit. Windows 32-Bit will add support for "opaque" file names, which will offload details of creating and parsing pathnames from the application to the operating system. Among the benefits is transparant support for accessing files on new file systems such as mini or mainframe servers sharing named resources on a network. The application is freed from knowing the file formats and network naming conventions when interacting with other PCs, VAX systems, or other "foreign" file systems. Removing hard-coded buffer sizes is the first step in taking advantage of opaque file naming.

Compile all sources at warning level 2 (-W2), warning level 3 (-W3) is recommended. Warning level 3 has been a problem in the past because WINDOWS.H included non-ANSI C compliant bitfield definitions that did not pass at this level. The latest release of the Microsoft C compiler (C 6.00a) moves this fatal error to -W4, allowing the strict type checking of -W3.

Create function prototypes for all functions. Relying on default C compiler handling is often (but not guaranteed to be) portable. In addition to parameter assumptions, the Microsoft C

compiler supports various calling conventions (_cdecl, _pascal, and so on), and the default calling convention may change because of future ANSI C requirements. Using function prototypes helps isolate source code from default compiler behavior and changes in the ANSI C definition.

Until recently, the size of WINDOWS.H has been a problem for the Microsoft C compiler, causing out of heap space problems in Pass 1 and/or Pass 2 of the compiler. This problem is corrected in the MS-DOS extender version of the MS C compiler (C 6.00ax). ISVs have worked around this previous limitation by extracting specific WINDOWS.H definitions into their source code. This could cause portability problems if these WINDOWS.H definitions are not updated with Windows 32-Bit definitions when the source is compiled under Windows 32-Bit. Therefore, either remove extracted header information and rely on WINDOWS.H, or clearly highlight extracted information for modification when building a Windows 32-Bit version.

4. Summary of Compatibility Rules

Rules for Windows 3.x Binary Compatibility on Windows NT

- Ensure that Windows 3.x applications run in Standard/Enhanced mode.
- Use published Windows 3.x APIs, messages, and structures.
- Do not modify WIN.INI directly; use a profile string API.
- Restrict direct port I/O (only standard devices).
- Do not directly access the disk controller; doing so is a security violation.
- Do not modify the system date and time; doing so is a security violation.

Rules for Portable Windows 3.x/Windows 32-Bit Source Code

- Parse *wParam* and *lParam* immediately in **WndProc** routines.
- NULL is a valid return value from **GetFocus** and **GetActiveWindow**.
- Use **FindWindow** instead of **hPrevInstance** to find other running instances.
- **GlobalLock** and **malloc** will not return 64K aligned pointers.
- Use Windows 3.x DIB functions to initialize color bitmaps.
- Do not use **GetInstanceData;** replace with supported IPC mechanism.
- Do not share GDI object handles (e.g. pens, bitmaps) between processes.
- Compile warning level -W2 or higher (-W3 recommended).
- Create function prototypes for all functions.
- Review structure member alignment and data types.
- Remove hardcoded buffer sizes (for example, file names and path names).
- Do not extract private copies of WINDOWS.H definitions.
- Use unique typdefs (HPEN, HWND, not HANDLE or int).