# The Win32™ Application Programming Interface: An Overview

Microsoft® WINDOWS™
SOFTWARE DEVELOPMENT KIT

# The Win32™ Application Programming Interface

## An Overview

Microsoft Corporation

Document No. 30214

# The Microsoft Windows Graphical Environment

Part

1

Since its original release in 1985, the Microsoft® Windows™ graphical environment has become the leading graphical system for personal computers. Microsoft Windows version 3.0, released in May 1990, was a milestone that broke the 640K barrier of the Microsoft MS-DOS® operating system by running applications in protected mode, thus making it possible to develop much more sophisticated applications. This innovation spawned myriad applications and is responsible for the huge success of the Windows environment in the marketplace, showcased by the volume of graphical applications sold (see Figure 1).

**Market Perspective**

| | 12/91 Installed base | Forecast annual run rate 1992 [1] | YTD 1991 application volume [2] |
|---|---|---|---|
| **Windows 3.0** | **7.9M** | **9.2M** | **$711M** |
| **MS-DOS** | **96.0M** | **24.3M** | **$2,148M** |
| **Macintosh** | **6.5M**[1] | **2.2M** | **$457M** |
| **PC UNIX** | **1.0M**[1] | **.4M** | **n/a** |
| **OS/2** | **1.2M**[1] | **.7M** | **$29M** |

**Sources: 1) IDC, October 1991; 2) Software Publishing Assoc., September 1991**

**Figure 1**

Between May 1990 and October 1991, more than 7 million personal computer users worldwide licensed Windows version 3.0. International Data Corporation estimates that an additional 9.2 million users will adopt it during 1992. In addition, more than 70,000 Microsoft Windows version 3.0 Software Development Kits (SDKs) have been shipped, a clear indication of the number of applications likely to appear during the next 12 to 18 months. By fall 1991, more than 5000 Windows-based applications were shipping.

Building on this achievement and on the success of independent software developers, Microsoft is extending and expanding the Windows environment so that Windows-based applications can run on a broad range of computing platforms—from battery-operated portables to high-end RISC workstations and multiprocessor servers.

We are expanding Windows to make it fully 32 bit and are adding additional operating system services. Microsoft Windows for Pen Computing and Microsoft Windows with Multimedia Extensions will also take advantage of new hardware technologies.

## Windows Today

Today many people think of Windows as a graphical add-on to the familiar MS-DOS operating system they have used for years. This perception took much of the fear out of upgrading to Windows for the end user, but, in fact, Windows is not limited by MS-DOS.

Windows is a complete operating system that provides extra features on top of MS-DOS and replaces certain MS-DOS features. Windows version 3.0 does not use MS-DOS screen or keyboard I/O, does not use MS-DOS memory management, and can even bypass MS-DOS file I/O with new Windows-specific device drivers. Windows version 3.0 Enhanced mode can handle 32-bit device drivers that are not limited by the infamous 640K MS-DOS barrier. These drivers talk through Windows to applications that are also not limited by the constraints of MS-DOS.

Because Windows works with MS-DOS, the value added by the MS-DOS long life span (in computer years) is preserved. Windows can run with MS-DOS TSRs and with MS-DOS device drivers, and, of course, it can run MS-DOS applications. Future versions of Windows will continue to be available on MS-DOS.

## The Windows Architecture

Since the IBM® personal computer was introduced in 1981, personal computers have become much more diverse in capability and in configuration. This diversity will increase in the next few years as personal computers based on RISC processors and multiprocessor systems are introduced.

These diverse systems have different operating system needs. For example, a battery-operated portable requires minimal memory and hard disk footprint to minimize weight and cost. It also requires power management to extend battery life. In contrast, network servers and mission-critical desktops require sophisticated security to ensure the integrity of data. Easy migration to RISC-based systems requires portability for both the operating system and the applications.

Some vendors feel that the diverse range of hardware requires totally different operating systems with incompatible applications. They sell different operating systems for personal computers, workstations, servers, and, in the future, pen-based systems. Each of their operating systems requires unique, incompatible applications. Connectivity between these divergent platforms is complicated.

Microsoft is focused on a much simpler solution. We're extending Windows into multiple, fully compatible implementations. Different implementations of Windows will be optimized for different classes of hardware. Customer investment in development for Windows and applications for Windows will be protected. Applications for Windows will run across the spectrum of hardware, from notepad-sized pen systems to mission-critical desktops to multiprocessor and RISC-based workstations.

Microsoft Windows is evolving into a complete operating system architecture that addresses diverse requirements by supporting different modes of operation. Today Windows has three modes: Real, Standard, and Enhanced. Real mode provides compatibility with previous versions of Microsoft Windows. Standard mode is optimized for an 80286 processor and provides access to the full 16 MB of memory supported by that chip. Enhanced mode takes advantage of the 80386 and 80486 processors by providing support for multiple simultaneous MS-DOS applications and, through a technique called demand paging, provides applications with access to more memory than is physically present in the machine. All three modes support both MS-DOS and Windows applications.

Building on the success of Windows version 3.0, Microsoft will introduce Windows version 3.1 in early 1992. Windows version 3.1 incorporates significant customer feedback. It includes TrueType™, an advanced scalable fonts technology; it improves performance, introduces a newly designed file manager, improves network connectivity, and improves system reliability. Windows version 3.1 will support Windows Standard mode and Enhanced mode.

Microsoft has also enhanced Windows Standard mode and Enhanced mode by providing extensions for sound, animation, and CD-ROM access, called Windows with Multimedia Extensions. In early 1992 we will release an operating environment for clipboard and pen-style computing, called Microsoft Windows for Pen Computing.

## Windows NT

In 1992, Microsoft will introduce a new product called Windows NT™ (New Technology). Windows NT is built on a 32-bit operating system kernel. Windows NT will deliver an extremely robust client environment for mission-critical applications, a high-end desktop platform, and a portable, scalable server environment (see Figure 2). Windows NT will also transform Windows into a Microsoft LAN Manager server platform, thus adding a fourth server platform to the three that LAN Manager currently supports: OS/2®, UNIX®, and VMS®.

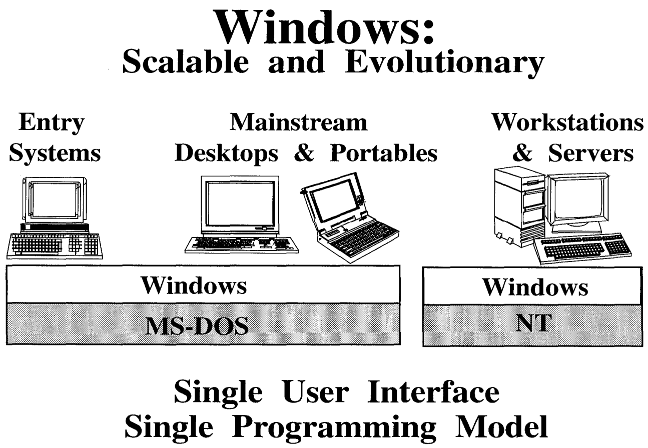

**Figure 2**

Windows NT does not require MS-DOS. It is, however, compatible with the large installed base of MS-DOS and Windows applications. In addition to providing compatibility with these existing applications, Windows NT includes the features required to meet the needs of the high-end desktop and server marketplace in the 1990s and beyond.

To support large server applications, Windows NT provides symmetric multiprocessor support, with threads symmetrically distributed among processors. This design provides maximum utilization of each processor in a multiprocessor system and simplifies the development of multiprocessor applications.

Network servers and many mission-critical applications require security. To meet this need, Windows NT has been designed as a secure operating system. Microsoft is working with the U.S. government to certify Windows NT as "C2-level" secure. In addition, the internal design of Windows NT can be enhanced in future releases to "B-level" security.

Windows NT is a key component of the Advanced Computing Environment (ACE) initiative announced in April 1991 by Microsoft, Compaq Computer Corporation, Digital Equipment Corporation, MIPS Computer Systems, The Santa Cruz Operation, and others. It now includes more than 200 members.

The goal of the ACE initiative is to provide an open, standards-based advanced computing environment for microprocessor-based systems as they become increasingly powerful. The ACE initiative provides full support for two platforms: 386/486-based PCs and MIPS RISC-based systems. As a portable operating system that spans both of these environments, Windows NT is a crucial element of the ACE standard. With Windows NT, existing MS-DOS and Windows programs will run unchanged on MIPS-based computers.

In addition to these advanced capabilities, the kernel-based design of Windows NT can be thought of as a nucleus that is compatible with different operating system environments. It exposes the Win32 API that is designed to support both client and server applications. In addition, the kernel design provides Windows NT compatibility with MS-DOS and Windows applications. It also provides the foundation for Windows NT to support OS/2 and POSIX subsystems, both of which are under development at Microsoft and will be available as add-on products.

## The Win32 API

Developers and end users have made enormous investments in programming for Windows and applications for Windows. Most of these applications have been developed to run on both the 16-bit 80286 processor and the 32-bit 80386 and 80486 systems. Although highly capable, programs written to the Windows versions 3.0 and 3.1 16-bit API are constrained by the memory limits inherent in a 16-bit architecture. Code must be divided into segments that cannot exceed 64K, which makes programming more difficult. Also, 32-bit code lets applications take full advantage of high-performance 80486 and RISC-based systems.

The Win32 API has been designed to make the transition from the Windows 16-bit API to 32 bit as easy as possible. Only minimal changes have been made to the syntax of the Win32 API. The API names are the same as those in Windows versions 3.0 and 3.1. The semantics and the message order are identical. In fact, it is possible to keep a single source code base and compile that source code into both 16-bit and 32-bit programs. The changes that are necessary are detailed in Part 2, "Portable Programming Considerations for Win32 Operating Systems."

Although the Win32 API is extremely compatible with the Windows 16-bit API, it also contains significant new features. These features include preemptively multitasked processes that use separate address spaces, preemptive threads, semaphores, named shared memory, named pipes, mailslots, and memory-mapped file I/O. Graphic device interface (GDI) improvements include Bézier curves, paths, and transforms.

The Win32 API will be fully supported in both MS-DOS Windows and Windows NT. The Win32 API will first be available in the Windows NT product during 1992. It will be added to MS-DOS Windows in 1993. Programs that are written to the Win32 API will have the same executable run on both Windows NT and MS-DOS Windows. All Win32 features will be supported by both MS-DOS Windows and Windows NT, including preemptive multitasking. Win32 programs will be fully source compatible between *x*86 and MIPS processors. SDKs for the Win32 API have been available to a select set of software developers since October 1991. SDKs will be generally available in the first half of 1992.

In addition, Microsoft Languages is developing a Windows extender product that will allow the creation of a 32-bit application that will run on both Windows version 3.1 and Windows NT. It does this by implementing a compatible subset of the Win32 API on Windows version 3.1. This product will provide a 32-bit programming environment for all Windows version 3.1 features but will not include advanced capabilities present in the full Win32 API such as preemptive multitasking. Additional information on this product will be released during the first half of 1992.

The following sections highlight some key features of the Win32 API.

## Kernel: The Base Operating System

The Win32 API on both Windows NT and MS-DOS Windows provides preemptive, thread-based multitasking. It also runs all Win32 and MS-DOS applications in separate address spaces so that they cannot corrupt one another or the operating system.

The Win32 API is designed to be portable beyond the 80386 and 80486 processors and in particular to be portable to RISC architectures. All these processors have different features but have in common 32-bit addressing and paged virtual memory architectures. Paged virtual memory is more efficient to implement and executes faster than segmented virtual memory. Memory management in Win32 is secure because the operating system places different memory objects in different pages of memory and allows an application to control access permissions (read, write, read-write, execute) to memory objects.

Win32 provides an API to allow an application to map files into its address space. Data within the file can then be accessed using simpler memory read-write instructions rather than I/O system functions such as rewind and seek. In addition, the operating system can conveniently and efficiently optimize file I/O done in this manner because of the large 32-bit address space available. The operating system, through page faulting, can detect read access to a file and bring in that data. It can detect when a shared file is written to and then write out that data. With process-configurable access permissions and sparse allocation of physical memory pages, processes can implement very efficient data access, even when access patterns are entirely unpredictable.

## GDI Improvements: Béziers, Paths, Transforms

GDI, the drawing API for Windows versions 3.0 and 3.1, provides a useful device-independent drawing set for applications. As output devices have become more sophisticated, so have drawing needs; hence, GDI has been improved.

Some applications for Windows versions 3.0 and 3.1 have needed to implement high-level graphics functions using the low-level drawing primitives of the Windows environment. Although this capability has provided application vendors flexibility in extending the Windows GDI, it has not allowed them to take seamless advantage of advances in printer and display technology. Application developers have had to code their own algorithms for displaying graphics such as Bézier curves and paths. With the Win32 API, developers can call new high-level graphics features that will take advantage of the built-in drawing capabilities in advanced output hardware. Under Win32, displaying Bézier curves can be handled by the graphics engine or by output devices that have implemented Bézier optimizations.

The Win32 GDI is a complete and general-purpose drawing package. Bézier curves are a curve primitive from which a straight line can also be derived. This function combined with the PolyBézier functionality makes it possible to draw any combination of continuous lines and curves.

Win32 also adds a **Path** API, making it easy for an application to manage multiple shapes efficiently. These shapes can consist of an arbitrary combination of lines, arcs, ellipses, and Bézier curves. A path is started by calling **BeginPath**. Subsequent calls to drawing primitives define the shape and size of the path. A call to

**EndPath** closes the path. Applications can then draw, clip through, fill, and transform these defined shapes.

The Win32 **Transform** API maps the virtual two-dimensional surface on which you draw to the two-dimensional output surface. This API, combined with the TrueType font technology first available with Windows version 3.1, makes it possible to draw truly device-independent graphics that the system can map to the display surface, including the rotation of bitmaps, fonts, and metafiles.

## The Windowing System and System Classes

The most significant change to the Windows windowing system is the desynchronization of the per-window message queue from the system message queue. This change prevents errant, looping applications that stop processing their messages from blocking the computer system's entire user interface and thus making other applications unavailable.

Desynchronization means that users can work with other programs while one application is busy. For example, if a word processing program is busy printing a 100-page document, a user can click another application's window or bring up the Task Manager to begin working in another application. This effectively minimizes the time the user waits with an hourglass on the screen.

The desynchronization of the message queue is completely compatible with the Windows versions 3.0 and 3.1 message models. The message ordering is the same. If WM_xyz came after WM_abc, it still does. This compatibility is necessary because, in Win32 systems, existing applications for Windows run on top of the Win32 message system. The messages are simply copied from the 32-bit stack to the 16-bit stack and passed to the application; therefore, message order cannot change.

## Networking Extensions

Each time the Windows API is extended to further standardize a particular area, it becomes easier to write significant new applications. Because of the variety of networking layers, ranging from network card interfaces and protocol stacks to the wide array of network interprocess communication (IPC) mechanisms, networking is probably the most confusing interface for developers today. Win32 will include a standard set of network APIs that can replace those that network providers have previously needed to supply. Win32 will expose driver-style interfaces similar to the **WinNet** API provided by Windows version 3.0 so that third-party vendors can plug their network services into the Windows open architecture.

Some of the new, 32-bit network APIs being defined are file, print, named pipes, mailslots, server browsing, and machine configuration. This means applications can rely on a consistent programming interface regardless of the underlying network. Even if a network is not present, the APIs are still available and will return appropriate error codes.

The Win32 API includes peer-to-peer named pipes, mailslots, and APIs to enable remote procedure call (RPC) compilers. With Win32, a mail-server vendor can build a messaging service on named pipes and asynchronous communication that will run on top of any network operating system, protocol stack, or network card—each of which could come from a different network vendor.

## Compatibility with the Windows 16-Bit API

Applications for Windows versions 3.0 and 3.1 will run on MS-DOS Windows and Windows NT. To be compatible with Windows versions 3.0 and 3.1, all 16-bit applications for Windows will run as one process in one address space. They will be nonpreemptive with respect to one another but preemptive with respect to the rest of the system, which mirrors their behavior under Windows versions 3.0 and 3.1 Enhanced mode. Applications for Windows run against the Win32 API without a "layer" and without any state mapping or message reordering.

Windows executables will also run on RISC-based Windows NT machines (see Figure 3). Excellent performance is expected on this platform because, although some code will be run against 80286 emulation technology, all Windows calls will be mapped directly to Windows NT calls and executed as native 32-bit instructions.

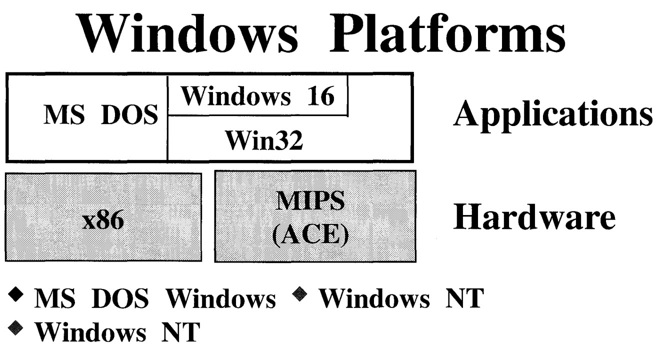

Figure 3

## The Future of Windows

Millions of people are actively using Windows version 3.0 today. Corporations and independent software vendors are making major investments in Windows and applications for Windows.

To protect this investment, Microsoft is evolving Windows into a complete architecture. Through separate implementations, Microsoft Windows will run on vastly different types of hardware, from pen-based notepad computers to multiprocessor and RISC systems.

Windows NT and future versions of MS-DOS Windows will support the Win32 API. Designed to simplify migration of applications for Windows from 16 bit to 32 bit, this API will also make it easy to develop new Win32 applications. It contains significant new features that will enable a new generation of more powerful applications for Windows.

In addition, the Win32 API will be used as the foundation for future versions of Windows under development at Microsoft. This technology, often called information at your fingertips, will make it even easier to use personal computers and will again provide significant new functionality to Windows users.

# Portable Programming Considerations for Win32 Operating Systems

Part

2

The first beta Win32 Software Development Kit for Windows NT will be available soon. Developers can update application source code to take advantage of new Windows version 3.1 features, make changes that will result in a robust application for Windows version 3.1, and prepare the application for transition into the full 32-bit environment provided by the Win32 API. This section is not a call to start creating 32-bit source code but rather a highlight of the changes that will benefit updating application source code for Windows version 3.1 and Win32.

## Goals of the Microsoft Win32 API

The creation of the Win32 API focused on six goals:

1. Provide a 32-bit migration path for existing Windows-based applications.

2. Make porting a Windows-based application to Win32 as easy as possible.

3. Create an efficient mapping layer to run Windows version 3.*x* binaries on Win32 systems.

4. Support a single source code base for creating Windows version 3.*x* and Win32 binaries.

5. Offer an identical Win32 API on both Windows NT and a future release of MS-DOS Windows.

6. Add a new API for advanced operating system features such as preemptive multitasking, IPC mechanisms, sophisticated memory management, and graphics capabilities while maintaining compatibility by simply widening the existing Windows API.

To achieve these goals, Microsoft derived the Win32 API from the existing Windows version 3.1 API, disallowing arbitrary name changes of data types, functions, and structures. At first glance, an application for Win32 is indistinguishable from an existing application for Windows version 3.0 or 3.1 (hereafter referred to as Windows version 3.*x*), both from a user's perspective and from a quick inspection of the source code. A native application for Win32 (unlike its cousin, which uses the Windows version 3.*x* API) can take full advantage of large linear memory allocation, multiple threads for background tasks and calculations, local and remote IPC via named pipes, and other features detailed in Part 1, "The Microsoft Windows Graphical Environment."

The Win32 API first appears in Windows NT for uniprocessor and multiprocessor 80386 and 80486 systems and for new RISC-based systems. A future version of MS-DOS Windows will also support the Win32 API. All Win32 features are supported by both Windows NT and the future release of MS-DOS Windows, including linear address space, threads, and preemptive multitasking. Win32-based applications running on MS-DOS Windows or Windows NT will be binary compatible with Intel® 80386 and 80486 processors and source compatible with Windows NT running on RISC processors.

This section concentrates on two aspects of Windows application portability:

- Steps that developers can take today while working on Windows version 3.1 applications to better support binary compatibility of these applications on Windows NT

- Techniques that developers can use to create Windows code that is more portable and that will make it easier to create Win32 versions of the application when the Win32 Software Development Kit for Windows NT is available

## Binary Compatibility

Win32 systems will be able to run existing Windows version 3.*x* applications with interoperability by means of dynamic data exchange (DDE), object linking and embedding (OLE), metafiles, and the Clipboard with other Windows version 3.*x* applications and with native Win32 applications. Applications for Windows version 3.*x* and applications for Win32 will exist side by side on the same display rather than running in separate screen groups. Applications for Windows version 3.*x* will be fully compatible with Windows NT if developers follow these rules:

- Ensure that Windows version 3.*x* applications run in Standard/Enhanced mode.

- Use published Windows version 3.*x* APIs, messages, and structures.

- Do not modify WIN.INI directly; use a profile string API (for example, **WriteProfileString**).
- Use QUERYESCSUPPORT to determine whether particular printer driver escapes are implemented.

The ability to run Windows version 3.*x* binaries on Windows NT is not restricted to 80386 and 80486 systems; these binaries will also run on RISC-based Windows NT systems. This is accomplished with a very high-performance PC emulator and the same efficient mapping layer technology used to seamlessly integrate applications for Windows version 3.*x* on Windows NT systems running on 80386 and 80486 systems.

## 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 Microsoft Windows Libraries for OS/2 (WLO) required special run-time 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 the product. This approach is unacceptable on Win32 systems.

To achieve binary compatibility and high performance on Win32 systems, developers of applications for Windows version 3.*x* do not need to recompile the source code, use special run-time libraries, or develop or acquire special tools to make executables compatible.

The ability to run Windows version 3.*x* binaries allows a user to update to Win32 systems and continue to use existing applications for Windows version 3.*x* as well as native Win32 applications as they become available. This protects investments in existing applications for Windows version 3.*x* and allows users to update to new Win32 applications as they are released. Native Win32 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 current products for Windows version 3.*x* on prerelease versions of Windows NT by means of a Windows NT beta test program to assure that binary compatibility is thorough and efficient.

## Supported Features

The following is a list of the many Windows version 3.*x* features supported on Win32 systems. It shows that existing applications for Windows can be binary compatible with future Win32 systems with little work on the part of developers. It also illustrates that complex windowing, graphics, and low-level operating system reliance by Windows version 3.*x* binaries will be completely supported.

Examples of major user interface features that are fully supported (no modifications needed) include:

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

Major graphical interface features that are fully supported include:

- TrueType and TrueType APIs
- Windows version 3.*x* icons and cursors in existing format
- Bitmaps (BMPs) and device-independent bitmaps (DIBs)
- Printing by means of native Win32 printer drivers

Base system functionality includes support for:

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

### Methods to Achieve Binary Compatibility

The Win32 API will employ a registration database that 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. The Windows version 3.*x* profile APIs should be used to manipulate all profile information. Installation programs that create private installation files should be modified to use the profile API or the Windows version 3.1 registration database API.

Windows NT includes a set of printer drivers similar to those in Windows version 3.*x*. This has been accomplished by sharing printer minidrivers. However, Windows NT also includes native Win32 printer drivers to take advantage of high-end printing capabilities present on such devices as PostScript® printers. Windows-based applications should always use QUERYESCSUPPORT before using any extended printer driver escape. Applications should not assume that printer drivers of a certain class (LaserJet® or PostScript, for example) are guaranteed to provide specific driver escapes. Querying for support guarantees that an application will not be affected by subsequent Windows version 3.*x* or Win32 printer driver updates.

Applications must be compatible with Windows version 3.*x* Standard or Enhanced mode. Win32 systems will not support Windows Real mode. Applications should use only published Windows version 3.*x* APIs, messages, and structures.

## Portable Coding Techniques

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

This discussion concentrates on the important issues that affect porting existing Windows source code to Win32. Although this list may seem long and detailed, all the recommendations are useful for creating robust Windows version 3.1 applications. In addition, applications will be more portable, and creating native Win32 applications will be easier when the Win32 Software Development Kit for Windows NT is available.

To assist in creating portable source code, the Win32 SDK will provide tools to automate the process. One tool is an editor with a table-driven search capability that can be used to search source code for APIs, messages, and certain C constructs that are nonportable. Once an item is found, the code is highlighted for review. Online help is available to assist in modifying source code.

## Source Code Rules

The following rules apply when writing portable Windows version 3.*x* or 32-bit source code:

- Parse *wParam* and *lParam* immediately in *WndProc* routines.
- Use portable API forms (for example, **MoveToEx** instead of **MoveTo**).
- NULL is a valid return value from **GetFocus** and **GetActiveWindow**.
- Use **FindWindow** (or IPC) instead of *hPrevInstance* to find other running instances.
- **GlobalLock** and **malloc** will not return 64K aligned pointers.
- Use Windows version 3.*x* DIB functions to initialize color bitmaps.
- Do not use **GetInstanceData**; replace with supported IPC mechanism.
- Do not share GDI object handles (for example, pens and 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 hard-coded buffer sizes (for example, filenames and paths).
- Do not extract private copies of WINDOWS.H definitions.
- Use unique typdefs (**HPEN**, **HWND**, not generic types such as **HANDLE** or **int**).
- Use portable integer typdefs (**UINT**, **WORD**).

## A Brief Look at Win32

If you start with Windows version 3.*x* source code, creating a native application for Win32 using the Win32 API is straightforward and requires minimal source changes. In general, the Win32 API simply involves widening parameters and return values to 32 bits. Over the course of a few months, a Windows NT development team ported a range of Windows version 3.*x* source code to Win32, including the complete Windows version 3.0 and beta 3.1 SDK sample code and relatively complex Windows version 3.1 applets—Program Manager, File Manager, Cardfile, and so on. This porting effort validated the design of the Win32 API and proved that creating applications for Win32 quickly from Windows version 3.*x* sources is possible. The Windows SDK samples and system applet source code were modified to be fully portable, allowing Windows version 3.1 and Win32 binaries to be created from the same code base.

The Windows version 3.1 system applets contain more than 100,000 lines of source code. File Manager contains approximately 20,000 lines of code; yet within one day, it was compiling as a native Win32 application. Within a week, 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 *x*86 and RISC processors. Few changes to the original Windows version 3.*x* C code were required, which is indicated by the short time needed to create a functional, portable version of File Manager.

An important porting factor is that Windows version 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 Win32: This is not surprising because the resource file is simply a script with no information that is 32-bit sensitive.

## Win32 Sample Source Code

The following code fragment is from the Windows version 3.0 SDK sample, GENERIC. Only one minor change to the entire GENERIC sample is required; the fragment builds completely as either a Windows version 3.*x* or Win32 binary. Although the GENERIC sample is not particularly sophisticated, it does contain a menu and a dialog box, indicating that more complex Windows functionality is easily supported.

If we look at the code fragment through the eyes of Win32, we see a true 32-bit application. Function parameters, pointers, and structure members are all widened from 16 to 32 bits. This widening is accomplished “under the covers” by means of typedefs in WINDOWS.H (Win32 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 from Windows Version 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 receiving message */
    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 subsequent 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 horizontal 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 */
}
```

## User Interface Code

This section details portability issues in user interface code. It includes issues related to message parameter packing, window and class extra words, profile string use, and localized input.

### Message Parameter Packing

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

WM_COMMAND

Win 3.*x*:

```
wParam == window id
lParam == hwnd, command
```

Win32:

```
wParam == window id, command
lParam == hwnd
```

The WM_COMMAND *window id* and *command* parameters remain 16-bit values in Win32 and can therefore be packed in the widened 32-bit *wParam.* The 32-bit *hwnd* value is now fully contained in *lParam.* Therefore, the notification code has been moved from the high word of *lParam* to the high word of *wParam.*

Code that tests for a message identifier should be modified:

Existing code:

```
switch (message) {
    :
    case WM_COMMAND:
        switch (wParam) {
            case ID_OK:
                :
        }
}
```

Portable code:

```
    switch (message) {
        :
        case WM_COMMAND:
            switch (LOWORD(wParam)) {
                case ID_OK:
                                                        :
                                    }
                }
```

In this case, the change can be made portably and continue to compile for either Windows version 3.*x* or Win32. **LOWORD**(*wParam*) extracts the correct low-order 16-bit message identifier in Windows version 3.*x* and Win32.

In extracting window handles from the WM_COMMAND message parameters, existing Windows code often uses constructs such as the following:

```
hwnd = LOWORD(lParam);
notification = HIWORD(lParam);
```

The portable method for extracting a window handle from the WM_COMMAND *lParam* is as follows:

```
hwnd = (HWND)(UINT)lParam;
```

**UINT** is a new data type discussed below. **UINT** casts *lParam* to a 16-bit value in Windows version 3.*x* (extracting the window handle) and a 32-bit value in Win32.

Handling the extraction of the WM_COMMAND notification code portably requires explicit coding:

```
#ifdef WIN32
    notification = HIWORD(lParam);
#else
    notification = HIWORD(wParam);
#endif
```

To minimize the effect of parameter packing differences, a set of macros that parse message parameters has been created. In this way, you can compile source code either as an application for Windows version 3.*x* or as an application for Win32 without unique message-handling code or C compiler **#ifdef** directives. Some programmers prefer macros; others prefer **#ifdef** statements. You can use both methods to create portable code. Examples of macros used to parse WM_COMMAND information are as follows:

```
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 version 3.*x* and Win32-specific, parsing the information from *wParam* or *lParam* as appropriate for each implementation. The important point is that you can easily create readable source code that can be compiled for Windows version 3.*x* or Win32.

## Summary

For messages that have changed their packing, extract the *wParam*/*lParam* information immediately upon handling the message. Use local variables to hold this information, and refer to the data using these variables—not by means of continued references but by means of *wParam* or *lParam* manipulations. Pass values extracted from *wParam* and *lParam*, not the *wParam* or *lParam* itself, to worker routines.

**LOWORD** and **HIWORD** are always suspect; verify each (search for them in an editor). Locating each occurrence will quickly highlight nonportable code. Study the target data type of these macros and the data type receiving the result.

You can use casts and/or macros to handle many porting issues.

Fortunately, there are few message differences, and most of the affected messages are used infrequently:

MESSAGE

Win 3.*x*: (Existing form)
    *wParam*: 16-bits
    *lParam*: Least Significant 16-bits, Most Significant 16-bits

Win32: (Widened form)
    *wParam*: Least Significant 16-bits, Most Significant 16-bits
    *lParam*: 32-bits

The following list can be used for quick reference. Similar approaches in handling packing differences with each message can be used as discussed for WM_COMMAND in the text above.

WM_ACTIVATE

Win 3.*x*:

```
    wParam: state
    lParam: fMinimized, hwnd
```

Win32:

```
    wParam: state, fMinimized
    lParam: hwnd
```

WM_CHARTOITEM

Win 3.*x*:

```
    wParam: char
    lParam: pos, hwnd
```

Win32:

```
    wParam: char, pos
    lParam: hwnd
```

WM_COMMAND

Win 3.*x*:

```
    wParam: id
    lParam: hwnd, cmd
```

Win32:

```
    wParam: id, cmd
    lParam: hwnd
```

WM_CTLCOLOR

Win 3.*x*:

```
    wParam: hdc
    lParam: hwnd, type
```

Win32:

```
    WM_CTLCOLORBTN
    WM_CTLCOLORDLG
    WM_CTLCOLORLISTBOX
    WM_CTLCOLORMSGBOX
    WM_CTLCOLORSCROLLBAR
    WM_CTLCOLORSTATIC
    WM_CTLCOLOREDIT
        wParam: hdc
        lParam: hwnd
```

**Note:** Porting WM_CTLCOLOR requires handling the specific control class color message. Portable code should use **#ifdef** directives to handle this difference.

WM_MENUSELECT

Win 3.*x*:
```
    wParam: cmd
    lParam: flags, hMenu
```

Win32:
```
    wParam: cmd, flags
    lParam: hMenu
```

WM_MDIACTIVATE (when message is sent to the MDI client window)

No change.

WM_MDIACTIVATE (when client window sends message to MDI child)

Win 3.*x*:
```
    wParam: fActivate
    lParam: hwndDeactivate, hwndActivate
```

Win32:
```
    wParam: hwndActivate
    lParam: hwndDeactivate
```

WM_MDISETMENU

Win 3.*x*:
```
    wParam: 0
    lParam: hMenuFrame, hMenuWindow
```

Win32:
```
    wParam: hMenuFrame
    lParam: hMenuWindow
```

WM_MENUCHAR

Win 3.*x*:
```
    wParam: char
    lParam: hMenu, fMenu
```

Win32:

```
wParam: char, fMenu
lParam: hMenu
```

WM_PARENTNOTIFY (also has two cases)

Win 3.*x* case #1:

```
wParam: msg
lParam: id, hwndChild
```

Win32 case #1:

```
wParam: msg, id
lParam: hwndChild
```

Win 3.*x* case #2:

```
wParam: msg
lParam: x, y
```

Win32 case #2:

```
wParam: msg
lParam: x, y
```

WM_VKEYTOITEM

Win 3.*x*:

```
wParam: code
lParam item, hwnd
```

Win32:

```
wParam: code, item
lParam: hwnd
```

EM_GETSEL

Win 3.*x*:

```
returns (wStart, wEnd)
wParam: NULL
lParam: NULL
```

Win32:

```
returns (wStart, wEnd)
wParam: lpdwStart or NULL
lParam: lpdwEnd or NULL
```

EM_LINESCROLL

Win 3.*x*:

```
    wParam: 0
    lParam: nLinesVert, nLinesHorz
```

Win32:

```
    wParam: nLinesHorz
    lParam: nLinesVert
```

EM_SETSEL

Win 3.*x*:

```
    wParam: 0
    lParam: wStart, wEnd
```

Win32:

```
    wParam: wStart
    lParam: wEnd
```

WM_HSCROLL:

WM_VSCROLL:

Win 3.*x*:

```
    wParam: code
    lParam: pos, hwnd
```

Win32:

```
    wParam: code, pos
    lParam: hwnd
```

## Window and Class Extra Words

The following APIs have nonportable implementations:

**GetClassWord**
**GetWindowWord**
**SetClassWord**
**SetWindowWord**

These APIs have two uses: to manipulate system information and to manipulate user-defined data. System data is modified by index values. The portability problem is that these APIs manipulate 16-bit data, but the data may need to widen to 32 bit on Win32. This is especially true for handle data accessed by means of predefined index values. For portability, these index values are now supported by means of the Windows extra long APIs:

**GetClassLong**
**GetWindowLong**
**SetClassLong**
**SetWindowLong**

The index values used to manipulate data that has widened to 32 bit are mapped as follows:

| | |
|---|---|
| GCW_CURSOR | —> GCL_CURSOR |
| GCW_HBRBACKGROUND | —> GCL_HBRBACKGROUND |
| GCW_HICON | —> GCL_HICON |
| GWW_HINSTANCE | —> GWL_HINSTANCE |
| GWW_HWNDPARENT | —> GWL_HWNDPARENT |
| GWW_ID | —> GWL_ID |
| GWW_USERDATA | —> GWL_USERDATA |

Therefore, to modify code that can compile in either Windows version 3.*x* or Win32, **#ifdef** directives are recommended:

```
#ifdef WIN32
hwndParent = (HWND)GetWindowLong( hWnd, GWL_HWNDPARENT );
#else
hwndParent = (HWND)GetWindowWord( hWnd, GWW_HWNDPARENT );
#endif
```

Alternatively, a named API already exists that obtains a parent Windows handle and is portable:

Nonportable:

```
hwndParent = (HWND)GetWindowWord( hWnd, GWW_HWNDPARENT );
```

Portable:

```
hwndParent = GetParent( hWnd );
```

Additional named APIs are being considered to address the remaining values only accessible by means of indexes in the current APIs. The named APIs will be available in Windows version 3.*x* and Win32.

The Windows extra word APIs are also used to manipulate user-defined data that may consist of private handles, pointers, or data that also must widen to 32 bit. Therefore, review all uses of these APIs in existing code to ensure that the data stored in Windows extra words remains 16 bit. Otherwise, use Windows extra long APIs to manipulate this data on both Windows version 3.*x* and Win32 even though the data may only be 16 bit in Windows version 3.*x*.

## Profile String Use

Win32 systems will 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 ensure 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.

Win32 versions of the Windows version 3.*x* profile APIs (for example, **GetProfileString** and **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.

## Localized Input

The Win32 model is different from Windows version 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 applications for Windows version 3.*x* and OS/2 Presentation Manager™, applications for Win32 will not be affected by other applications that process their messages slowly or that 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 )**
**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, 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 version 3.*x* semantics of APIs that return input-synchronized states are changed slightly. For example, an HWND and a return value of TRUE for success can call **SetFocus**, 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 version 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 functions.

Mouse capture is affected in an added dimension. The Win32 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 Win32 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 until 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.

## Graphics Device Interface Code

This section details changes that need to be made to an application's GDI code.

### Portable Solutions for Win32 API Changes

The Windows API consists of several hundred APIs; all are widened to 32 bits with minimal impact on existing source code, except for approximately two dozen, which are generally GDI related. Unfortunately, these specific APIs could not be supported as-is in the Win32 API but had to be modified. Most of these APIs fall into a specific class. Previously, they returned a packed X/Y value in a DWORD return value. Because graphics coordinates are now 32 bit rather than 16 bit, an extra parameter has been added to these functions, a pointer to a POINT structure. To simplify porting, the Win32 forms for these functions are included in Windows version 3.1. Windows version 3.1 will support the old and the new form. The Win32 API names are based on the original with an **Ex** suffix added. Win32 will support only the new form. If you write to the new form, you can compile for either Win32 or Windows version 3.*x*.

Windows version 3.1 will implement the API by means of a static library so that code compiled with the new API will also function in Windows version 3.0.

Modifying code to use the new forms of these APIs is straightforward in most cases. Half of the APIs are used to get coordinates from GDI. The other half set GDI coordinates and return the previously set value.

**MoveToEx**
**OffsetViewportOrgEx**
**OffsetWindowOrgEx**
**ScaleViewportExtEx**
**ScaleWindowExtEx**
**SetBitmapDimensionEx**
**SetMetaFileBitsEx**
**SetViewportExtEx**
**SetViewportOrgEx**
**SetWindowExtEx**
**SetWindowOrgEx**

In general, most Windows applications ignore the return value from the above functions. Therefore, modifying an existing application to use the new forms of these APIs is straightforward:

Original:

```
MoveTo( hDC, x, y );
```

Portable:

```
MoveToEx( hDC, x, y, NULL );
```

In cases where the return value is used, you must modify code to use a structure rather than using the packed DWORD in the original API. This is identical to the matching APIs that are used explicitly to obtain X/Y information.

**GetAspectRatioFilterEx**
**GetBitmapDimensionEx**
**GetBrushOrgEx**
**GetCurrentPositionEx**
**GetTextExtentPoint**
**GetTextExtentPointEx**
**GetViewportExtEx**
**GetViewportOrgEx**
**GetWindowExtEx**
**GetWindowOrgEx**

Of these APIs, **GetTextExtent** is the most common to be encountered. In the case of **GetTextExtent**, the **Point** suffix has been used because **GetTextExtent** already has a Windows version 3.1 **GetTextExtentEx** extended function. Therefore, the mapping of related functionality is as follows:

| | |
|---|---|
| **GetTextExtent** | **—> GetTextExtentPoint** |
| **GetTextExtentEx** | **—> GetTextExtentPointEx** |

Because the Win32 API form now relies on a structure, the typical coding change requires creating a temporary (local) structure:

Nonportable:

```
dwXY = GetTextExtent( hDC, szFoo, strlen( szFoo ) );
rect.left = 0; rect.bottom = 0;
rect.right = LOWORD(dwXY); rect.top = HIWORD(dwXY);
InvertRect( hDC, &rect );
```

Portable:

```
{
SIZE sizeRect;

GetTextExtentPoint( hDC, szFoo, strlen( szFoo ), &sizeRect );
rect.left = 0; rect.bottom = 0;
rect.right = sizeRect.cx; rect.top = sizeRect.cy;
InvertRect( hDC, &rect );
}
```

Another class of necessary Win32 changes concerns a handful of Windows APIs that do not provide a parameter for specifying sizes of buffers receiving data. This is an API design error that is corrected in Win32. The functions affected are as follows:

**DlgDirSelectEx**
**DlgDirSelectComboBoxEx**

And the changes for portability are minor:

Win3.*x*:

```
DlgDirSelect( hDlg, lpString, nIDListBox );
```

Portable:

```
DlgDirSelectEx( hDlg, lpString, sizeof( lpString ), nIDListBox );
```

As noted previously, all functions listed in this section have equivalent, portable versions in Windows version 3.1. Therefore, converting to the Win32 forms is not a one-way street.

## DIBs vs. DDBs

Beginning with Windows version 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 version 3.*x* and Win32. Device-dependent bitmaps (DDBs), as originally offered in Windows versions 1.*x* and 2.*x*, are not recommended.

Because DDBs lack complete header information, applications that directly manipulate DDB data are not portable to Win32. Developers are encouraged to write to the Windows version 3.*x* DIB APIs (for example, **SetDIBitmapBits**); these calls are portable. Win32 does provide, however, a subset of functionality for DDB APIs, 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. In low memory situations in Windows versions 1.*x* and 2.*x*, the bitmap in memory could be freed 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. Although caching is not needed in Windows version 3.*x* or Win32, source code employing this technique will still be supported.

Win32 does not support initializing color DDBs with **CreateBitmap**. Such code is also not portable among Windows version 3.*x* systems with different display drivers because DDB data is device-driver dependent.

## Sharing Graphical Objects

All applications for Windows version 3.*x* run 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.

Applications for Win32 run in separate address spaces, and graphical objects are owned by the process that creates them. Only the owner of a graphical object can manipulate it. 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 (by means of standard IPC) and create its own copy of the bitmap. Alterations to the bitmap must be communicated between the cooperative processes by means of IPC and a proper protocol. One such protocol is DDE. Win32 may add an explicit ownership transfer API for graphical objects to allow cooperative applications to share graphical objects.

## Base System Support

This section details portability issues for base Windows functions, including instance initialization, memory allocation, **INT21** functions, and DLLs.

### Instance Initialization

The first release of Windows (version 1.01) was designed to run on 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 protected systems in which applications run in separate address spaces, these functions are no longer appropriate.

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

A Win32 version of **WinMain** supports the same parameter list as does Windows version 3.*x*:

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

However, the *hPrevInstance* parameter 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 version 3.*x*, multiple instances can share private window classes registered by the first instance. Under Win32, each instance is required to register its own window classes.

Applications usually test *hPrevInstance* to see if they must register their window class. This test is guaranteed to work optimally under Win32, always indicating the first instance of the application, and Win32 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 ensuring 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 Win32 to test for previous instances. These applications must use an alternative method, such as creating a unique named pipe, creating/testing for a named semaphore, broadcasting a unique message, or calling **FindWindow**. If another instance is found, the application determines which instance should be terminated.

## Memory Manipulation

Under the Windows version 3.*x* segmented memory architecture, globally allocated memory always aligns on segment boundaries. Both **GlobalAlloc** and the C run-time **malloc** family of functions allocate global memory in a way that causes the 16-bit offset of the 32-bit segmented pointer that references the base address of this data always to be 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. (Pointer manipulation is discussed later in this document.)

## Win32 API Replacements for INT21

Direct **INT21** functions or the use of the Windows version 3.*x* **DOS3Call** API to request MS-DOS to perform file I/O operations must be replaced by the appropriate Win32 file I/O calls. Win32 has a complete set of named APIs to replace nonportable **INT21** functions.

| **INT 21H Function** | **MS-DOS Operation** | **Win32 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 run-time libraries are sufficient for normal file I/O. The C run time has the advantage of being portable across many platforms.

## Dynamic Link Libraries

DLL initialization and termination functions behave differently in Windows version 3.*x* and Win32 in terms of how they are defined, when they are called, and the information that is made available to them. Win32 DLLs are easier to create and have functionality not currently available in Windows version 3.*x*. In Windows version 3.*x*, initialization and termination functions must be provided, the termination function must be named WEP, and the initialization function is the DLL entry point written in master. Initialization and termination functions are optional in Win32 DLLs.

In Windows version 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 called. 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 Win32, 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 version 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 Win32, DLLs implemented in assembly language should be rewritten in C so that they are portable to RISC-based systems.

Windows version 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

Win32 DLL initialization functions are passed the following information:

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

The Win32 module handle is analogous to the Windows version 3.*x* instance handle. In Win32, 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 Win32 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 because in Win32 it can be obtained through an API function.

Although the Windows version 3.*x* LIBENTRY.ASM function contains nonportable assembly routines, it isolates the assembly language initialization and supports writing additional DLL-specific initialization in C by means of the **LibMain** function. For portability to Win32, DLL initialization code should be added to the **LibMain** function and written in C. (For further information on LIBENTRY.ASM, see the Windows version 3.*x* SDK documentation.)

## C Coding Guidelines

The Win32 API was designed to simplify the creation of Win32 applications from Windows version 3.*x* sources. Specific API differences have been discussed above. Creating portable Windows code also involves writing portable C. Fortunately, the similarity of Windows version 3.*x* and Win32 requires only that a concise set of portable C guidelines be followed. Windows programs have generally been optimized to operate with the segmented *x*86 architecture. Therefore, the change from segmented to linear memory is the most significant issue in creating portable C code.

## Pointer Manipulation

Win32 supports a compatible set of memory management functions, such as **GlobalAlloc** and **GlobalLock**, and a new set of advanced linear memory APIs. Therefore, existing applications for Windows can easily be converted to Win32 and continue to use the Windows version 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 Win32. 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 version 3.*x* will grow from 2 bytes to 4 bytes in Win32.

## 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 Win32. 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 32,767 or 65,535 will experience problems when the integer loop counters grow to 32 bit and wrapping occurs at 2 GB or 4 GB.

## 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 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 Win32.

**Win32 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.

## 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 version 3.*x* and Win32, 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 Win32 than do generic typedefs such as **HANDLE**. Although all handles in Windows version 3.*x* are interchangeable with **HANDLE** or **unsigned int**, using these basic data types affects porting to Win32 because various objects require different typedefs under Win32 than under Windows version 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.

## UINT vs. WORD

Win32 relies on existing Windows version 3.*x* WINDOWS.H typedefs to automatically widen parameters and structure members to 32 bits as well as retain 16-bit data types for compatibility. All handles (HWND, HANDLE, HPEN, HBRUSH, and so on) grow to 32 bits. Data types such as **LONG** and **DWORD** are 32 bit in both Windows version 3.*x* and the Win32 API.

In creating the Win32 API, it was recognized that a new, flexible typedef was required. The **UINT** data type is defined to be an unsigned int. This data type is 16 bits in Windows version 3.*x* and 32 bits in Win32. Therefore, this is a portable data type that also takes advantage of 32-bit mode. Accessing 32-bit data is more efficient than accessing 16-bit data, especially for RISC processors. The **WORD** typedef is generally considered a 16-bit quantity and remains a 16-bit data type in Win32. **UINT** and **WORD** data types are defined as follows:

```
typedef unsigned int UINT;
typedef unsigned short WORD;
```

Therefore, **WORD** remains 16 bit on Win32 and should be used to specify only objects that should exist as 16-bit unsigned types. **UINT** should be used to define objects that naturally widen in 32-bit mode.

Structures and API prototypes in Windows version 3.1 header files have replaced **WORD** with **UINT** for structure members and API parameters that should widen to 32 bits. Although **UINT** and **WORD** are interchangeable (for compatibility) in Windows version 3.1 headers, they are not interchangeable in Win32. Therefore, applications should review the following typedefs and update source code to use the proper data types for portability:

Flexible data types:

| | |
|---|---|
| **int** | ; 16-bit signed integer on Windows 3.*x*, 32-bit signed integer on Win32 |
| **UINT** | ; 16-bit unsigned integer on Windows 3.*x*, 32-bit unsigned integer on Win32 |

Fixed-size data types:

| | |
|---|---|
| **WORD** | ; 16-bit unsigned integer in Windows 3.*x* and Win32 |
| **DWORD** | ; 32-bit unsigned integer in Windows 3.*x* and Win32 |
| **LONG** | ; 32-bit signed integer in Windows 3.*x* and Win32 |

## Portable Use of the WORD Data Type

Historically, Windows programs have used the various typedefs interchangeably: **WORD**, **HWND**, **HANDLE**, and so on. It is not uncommon to see (WORD) casts being used to assign values to variables holding handles to windows:

Nonportable:

```
HWND hWnd;
hWnd = (WORD)SendMessage( hWnd, WM_GETMDIACTIVE, NULL, NULL );
```

Portable:

```
HWND hWnd;
hWnd = (HWND)SendMessage( hWnd, WM_GETMDIACTIVE, NULL, NULL );
```

In porting the Windows version 3.*x* system applets, games, and sample code, most (WORD) casts were found to be nonportable.

Review all (WORD) casts in existing Windows version 3.*x* code to determine the data type/size of the original value and result.

## 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 version 3.1 code and will create code that is more easily ported to Win32.

Review hard-coded buffer sizes for file names and environment strings. Although dynamically allocating buffers to hold strings is not necessary, Win32 supports FAT 8.3 and 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 Win32.

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 bit-field 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 C compiler implementations. 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 Microsoft 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 Win32 definitions when the source is compiled under Win32. Therefore, either remove extracted header information and rely on WINDOWS.H or clearly highlight extracted information for modification when building a Win32 version.

This overview has concentrated on the most common issues that will be encountered in creating a portable Windows application. Although a significant number of changes may appear to be required for portability, in practice, creating a portable application for Win32 systems (either native 32 bit or by means of binary compatibility) is straightforward. Porting tools help automate the process.

With the compatible changes being made in the Windows version 3.1 SDK, there is truly one Windows API with 16-bit segmented and 32-bit linear forms. Although the Win32 API offers new advanced features, the semantics of the existing Windows API are not broken. Only a small percentage of APIs and messages were affected by 32-bit widening. In these situations new, portable solutions have been provided in the Windows version 3.1 API.

## Summary of Compatibility Rules

Rules for Windows version 3.*x* binary compatibility on Windows NT:

- Ensure that applications for Windows version 3.*x* run in Standard/Enhanced mode.
- Use published Windows version 3.*x* APIs, messages, and structures.
- Do not modify WIN.INI directly; use a profile string API.
- Use QUERYESCSUPPORT to determine whether particular printer driver escapes are implemented.

Rules for portable Windows version 3.*x*/Win32 source code:

- Parse *wParam* and *lParam* immediately in *WndProc* routines.
- Use portable API forms (for example, **MoveToEx** instead of **MoveTo**).
- 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 version 3.*x* DIB functions to initialize color bitmaps.
- Do not use **GetInstanceData**; replace with supported IPC mechanism.
- Do not share GDI object handles (for example, pens and 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 hard-coded buffer sizes (for example, file names and paths).
- Do not extract private copies of WINDOWS.H definitions.
- Use unique typdefs (**HPEN**, **HWND**, not **HANDLE** or **int**).
- Use portable integer typdefs (**UINT**, **WORD**).

Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-6399

Microsoft®

0392 Part No. 30214