Introduction to Microsoft Win32s

Microsoft® Win32s® version 1.25 is an operating system extension that allows Win32® applications for Microsoft® Windows NT™ and Microsoft® Windows® 95 to run on Microsoft® Windows® version 3.1*x* and Microsoft® Windows® for Workgroups.

Win32s offers software developers:

- A 32-bit programming model for Windows 3.1 that shares binary compatibility with Windows NT and Windows 95.
- Full OLE support, including 16-bit/32-bit interoperability.
- Performance advantages of 32-bit mode.
- Win32 semantics for the application programming interface (API).
- A rich subset of the full Win32 API found on Windows NT and Windows 95.
- An established market of Windows 3.1*x* systems and the new Windows NT and Windows 95 market to sell Win32-based applications.
- The ability to ship a single Win32 product for Windows NT, Windows 95, and Windows 3.1*x.*

Win32s consists of a virtual device driver (VxD) and a set of dynamic-link libraries (DLLs) that extend Windows 3.1*x* to support Win32-based applications. The Win32s files must be shipped with the Win32-based application and installed on the Windows 3.1*x* system.

The Microsoft Win32 Software Development Kit (SDK) provides the following Win32s components:

- Win32s binaries and Setup program to install Win32s on a Windows 3.1*x* system
- Development files as part of the Win32 SDK to create, debug, test, and ship your Win32-based application and Win32s files.

The Win32 SDK relies on Windows NT- or Windows 95 - hosted development tools, such as the compiler and debugger, to build and test your Win32-based application.

Win32s is a licensed technology and a number of compiler vendors provide Win32 development solutions on MS-DOS®, Windows 3.1*x*, Windows NT, and Windows 95 platforms.

This section provides Win32s installation instructions using a setup program provided on the Win32 SDK CD-ROM. The rest of this documentation covers issues developers should consider when writing Win32-based applications targeted for both Windows NT and Windows 3.1*x*.

New Features

In Win32s version 1.25, the 128K stack limitation of previous releases of Win32s has been removed. Support has been improved for Stacker® and Novell® Netware®.

Win32s System Requirements

To install Win32s and run Win32-based applications, you need:

- A system running Microsoft Windows for Workgroups or Windows 3.1*x* in enhanced mode with paging enabled (default installation).
- A 386, 486 or Pentium™ processor (386sx and 486sx processors are supported).
- 6MB memory requirement (4MB for Win32 core and 2MB for OLE).
- 2.2MB of hard disk space for Win32s and another 2MB for OLE (for a total of 4.2MB).

Installing Win32s

The Win32s system files can be installed from the Win32 SDK CD-ROM, from a shared CD-ROM drive over the network, or from a 3.5-inch or 5.25-inch high density floppy disk to which you've copied the system files from the Win32 SDK CD-ROM (there is no guarantee that the 5.25-inch floppy disks will match). In all cases, the same setup program is used to install and enable Win32s on a Windows 3.1*x* system. Please note that installing Win32s requires write access to the Windows system directory, which may require special permissions on a networked installation.

Making a Floppy-Disk Version of Win32s

The Win32s files are located on the Win32 SDK CD-ROM in the directory \MSTOOLS\WIN32S\DISKS. Making a floppy-disk version of Win32s requires either an MS-DOS or Windows NT system that can access a CD-ROM drive directly or over a network, and two or three floppy disks (two floppies for the Win32 core and three floppies for the Win32 core plus OLE).

1. Place a blank, formatted disk (3.5-inch) in your floppy drive.
2. Copy all files in \MSTOOLS\WIN32S\DISKS\RETAIL\OLE32S\DISK1 into the root directory of the floppy disk.

   If you are installing the Win32s core only, use this path: \MSTOOLS\WIN32S\DISKS\RETAIL\WIN32S (which requires two floppies only).
3. Place another blank, formatted disk (3.5-inch) in your floppy drive.
4. Copy all files in \MSTOOLS\WIN32S\DISKS\RETAIL\OLE32S\DISK2 into the root directory of the floppy disk.
5. Place another blank, formatted disk (3.5-inch) in your floppy drive.
6. Copy all files in \MSTOOLS\WIN32S\DISKS\RETAIL\OLE32S\DISK3 into the root directory of the floppy disk.

Installing Win32s from a Floppy Disk

1. Start your computer, then start Windows (type **win** at the MS-DOS prompt).
2. Place the Win32s Setup Disk1 in your floppy drive.
3. In File Manager, display the files on the floppy, then double-click on SETUP.EXE to run the Setup program.
4. Follow the setup instructions to complete the installation.

Installing Win32s from CD-ROM or a Network

1. Start MS-DOS with appropriate software for accessing a local CD-ROM drive or a remote CD-ROM drive over the network.
2. Start Windows (by typing **win** at the MS-DOS prompt).
3. Use File Manager to access the Win32 SDK CD-ROM.
4. In File Manager, display the files in the \MSTOOLS\WIN32S\DISKS\OLE32S\DISK1 directory, then double-click SETUP.EXE to run the Setup program.

   If you are installing the Win32s core only, use this path: \MSTOOLS\WIN32S\DISKS\RETAIL\WIN32S
5. Follow the setup instructions to complete the installation.

Verifying Win32s Installation

In addition to installing the Win32s system components, the Win32s Setup program gives you the option to install the Win32 card game FreeCell, the same program that ships as part of Windows NT. After installing Win32s, you can run FreeCell to verify that Win32s is installed correctly. FreeCell is located in the Program Manager group Win32-based applications, which the Win32s Setup program creates.

A common source of problems with Win32s is a bad display driver. Older versions of the S3 display drivers are incompatible with Win32s, and when a Win32 application (such as FreeCell) is run, errors ranging from display corruption to system hangs are reported. Other video cards that do not use S3 accelerator chips have been reported to have similar problems. To obtain newer drivers, contact your display card manufacturer. Until you get drivers that work you can switch to the VGA or SuperVGA drivers included with Windows.

Note If you are installing Win32s on Windows 3.1*x*, you must run either the MS-DOS Share or VSHARE.386 utility before starting Windows. Either add SHARE.EXE to your AUTOEXEC.BAT file or install VSHARE.386 in SYSTEM.INI by adding the line "device=VSHARE.386" to the [386Enh] section. Windows for Workgroups installs VSHARE.386 by default, and does not need SHARE.EXE.

Disabling Win32s

You shouldn't need to disable Win32s. The Win32s DLLs will be loaded only when a Win32-based application is executed. The Win32s VxD is loaded when Windows starts, but has little memory overhead. If you must disable Win32s or wish to reinstall Win32s, take the following actions:

- Remove the Win32s VxD line from Windows SYSTEM.INI file in the [Enh386] section:

  ```
  device=<SYSTEM>win32s\w32s.386
  ```

- Modify the following line from the [BOOT] section in the SYSTEM.INI file:

  ```
  drivers=mmsystem.dll winmm16.dll
  ```

  to the following (remove WINMM16.DLL, leaving any other DLLs on the line):

  ```
  drivers=mmsystem.dll
  ```

- Delete the WIN32S.INI, W32SYS.DLL, WIN32S16.DLL, and WINMM16.DLL files from the <SYSTEM> directory and all files in the <SYSTEM>\WIN32S subdirectory.

  <WINDOWS> is the Windows installation directory, such as C:\WINDOWS. <SYSTEM> is the Windows System directory, which may be on a networked drive and not a subdirectory of the <WINDOWS> directory. If you are removing only Win32s from your computer, you do not need to remove the shared files (in <SYSTEM> and <SYSTEM>\WIN32S). Before removing these files from the network share, make sure that all users of Win32s have removed the references to Win32s from the SYSTEM.INI file.
- Restart Windows.

Technical Overview

This section provides architectural details and high-level functionality information on features supported by Win32s.

Features

The Win32s system extension allows Win32-based applications to make 32-bit calls to Windows 3.1*x*, which is then responsible for all graphics and windowing operations. The Windows API in Windows 3.1*x* and the Win32 API in Windows NT and Windows 95 have many common features.

The following list highlights Windows 3.1*x* features available through Win32s:

- Complete windowing interfaces (User)
- All graphics functions (GDI)
- OLE
- DDE/DDEML (dynamic data exchange and DDE Management Library)
- TrueType fonts
- Common dialogs
- Network support (NetBios and Windows Sockets 1.1 (*) APIs)
- Multimedia support (sound APIs)

In addition to supporting Windows 3.1*x* functionality, the following Win32 features are also supported through Win32s on Windows 3.1*x*:

- Structured exception handling (SEH)
- Sparse memory (Virtual memory API)
- Memory-mapped files (backed by disk image)
- Growable heaps (Heap API)
- International Support (localized versions of Win32s and Code Page/Unicode™ translation APIs)

The following new features have been added in Win32s 1.2:

- OLE support
- New National Language Support (NLS) functions
- Memory-saving compressed code page files
- Combined libraries that require less memory

Win32s provides a 32-bit Windows Socket translation layer and requires that your system already have 16-bit Windows Sockets 1.1 installed. A number of third-party vendors ship TCP/IP and Windows Sockets products for MS-DOS and Windows. Windows Sockets support for Windows for Workgroups is freely available.

Win32s offers a 32-bit solution for Windows 3.1*x* supporting binary compatibility with Windows 95 and Windows NT running on 386/486 hardware and source compatibility to RISC platforms such as the MIPS R4x00, DEC™ Alpha AXP™, and Motorola PowerPC architectures.

Architecture

The following figure illustrates the key pieces of Win32s. The dark gray boxes represent Win32s components; the light gray boxes depict Windows 3.1*x* components. The 32-bit application dynamically binds to W32SCOMB.DLL, which is the combined Win32s version of the Win32 system DLLs KERNEL32.DLL, GDI32.DLL, USER32.DLL, and others. A large percentage of the API calls are then handled by the general 32/16 thunking DLL, WIN32S16.DLL, which is responsible for repacking the stack, truncating or expanding parameters, mapping message parameters (due to Win16/Win32 differences), etc. The VxD is trusted ring 0 kernel code that handles low-level system services such as exception handling, floating-point trap emulation, and low-level memory management.

{ewc msdncd, EWGraphic, group10901 0 /a "SDKS-1.bmp"}

Because Windows 3.1*x* does not recognize and cannot load Win32 executables, Win32s provides Win32 loader code. Loader services are handled by W32SKRNL.DLL.

The W32SCOMB component contains the Windows 3.1*x* window manager, graphics engine, and kernel services DLLs. Figure 1 shows the general flow of control when a Win32-based application makes a Win32 call. Most calls are passed to Windows 3.1*x*, which actually implements the call.

Coordination Between Windows 3.1*x* and Win32s

Win32s and Windows 3.1*x* system-code are tightly coupled. Win32s is a true system extension to the Windows 3.1*x* operating system code.

The first level of coordination occurs in the Windows 3.1*x* loader. The Windows 3.1*x* loader was designed to recognize the presence of a Win32s loader. When Windows 3.1*x* attempts to load an executable and cannot identify the image header, the Win32s loader (if present) is called to determine whether the image is a Win32 executable, known as "PE," for Portable Executable. If the image is not a PE image, Windows 3.1*x* reports that file cannot be executed. If the image is a PE image, the Win32s loader loads the image table (fix-up tables, etc.) and then demand-loads the rest of the image as the program executes.

Win32s also hooks the Windows 3.1*x* resource loader, so that application icons appear correctly in the Program Manager, or other resource-viewing utilities. When Windows 3.1*x* attempts to read a resource from a PE image (something that it cannot do), Win32s fabricates the resource (such as the icon) as a 16-bit resource. In this way, Win32-based applications can be added to the Program Manager and still have properly displayed icons for selection. This support is transparent to both the programmer and the user.

Win32s uses a similar method to support version resources. Versioning APIs are new to Windows 3.1*x* and allow version-marked executables to be checked by utilities such as setup tools. Win32s hooks Windows 3.1*x* so that version calls return valid information when a 16-bit Windows application version-checks a Win32-based application.

Messaging between applications is generally handled in a straightforward manner by the Win32s thunk layer. However, special cooperation between Win32s and Windows 3.1*x* also allows Windows hook APIs and message subclassing to be supported. A Win32-based application can install a hook to monitor all Windows applications (16- and 32-bit). Win32s ensures that the Win32 hook sees a 32-bit format of all messages regardless of the message source or destination.

Windows 3.1*x* and Win32s also require cooperative memory management between the Win32s VxD and Win386. Win32s also manages the task of translating pointers passed via the Windows API between the Win32-based application and 16-bit Windows system code.

Handling Messages

Message handling in a Win32-based application is generally identical to message handling in an equivalent 16-bit Windows application. A Win32-based application receives messages in the same order as a Win16 application because Win32s relies on Windows 3.1*x* to deliver messages. Message order is also the same on Windows NT and Windows 95.

Memory

Windows 3.1*x* implements a shared memory design, in which all Windows processes reside in one global memory heap. Windows NT, however, isolates processes into their own, private address space. Win32-based applications running on Windows 3.1*x* coexist in the same shared global memory heap as other Win32- and Win16-based applications.

Win32s supports the following features for Win32 executables or DLLs:

- Multiple Win32 executables and DLLs can be loaded simultaneously.
- Multiple Win32 executables can reference the same Win32 DLL.
- Win32 DLLs referenced by multiple Win32-based applications share a single copy of the DLL code and data (just like 16-bit Windows DLLs).
- Win32s DLLs and VxD are shared by all Win32-based applications.
- Win32s supports executing data as code, such as dynamic code generation (**FlushInstructionCache** should be used where appropriate).
- Win32-based applications see a flat 32-bit address space (CS, DS, SS, and ES map a single address space).
- Win32s supports demand-paging of executables for efficient loading of executables, DLLs and data.
- Win32s allocates a growable stack for each Win32 process. Stack size is specified at link time (linker switch); Win32s will create a stack that is at least 64K.

Because Win32 executables share a global heap and cannot use selectors for dereferencing addresses (as do Win16 applications), multiple instances of Win32 executables do not share code. The loader must fix up all 32-bit linear addresses to code and data when an executable is loaded. Win32-based applications do not move in memory nor are segment registers available to switch context to a second instance while still sharing the same code. Therefore, each instance of a Win32 executable must be loaded into memory and code cannot be shared.

Note Win32-based applications should not be designed to assume that all Win32-based applications run in the same address space. Such applications will not run on Windows NT or Windows 95.

Processes and Tasking

Win16- and Win32-based applications coexist seamlessly on the same desktop (display). Win32s assures that Win16-based applications "see" Win32-based applications as other Win16-based applications, and that Win32-based applications see Win16-based applications as other Win32-based applications. A Win32-based application can enumerate windows in the system and will receive 32-bit window handles for both Win32- and Win16-based applications. This means that Win32-based applications can send messages to other windows or can use DDE or OLE to communicate with other processes without regard to the 16- or 32-bit nature of the other application.

The Windows 3.1*x* non-preemptive, message-driven scheduler is responsible for scheduling both Win16- and Win32-based applications. Win32-based applications operate under the same constraints as Win16-based applications. To keep the user interface responsive, Win32-based applications must check their message queue regularly. On Windows NT, a Win32-based application that does not check its message queue and handle messages in a timely manner only affects its own user interface and the user may switch away and work on other applications. Windows 3.1*x*, however, uses messages to schedule processes. Therefore, applications (16- or 32-bit) must handle messages in a timely manner in order to keep Windows 3.1*x* responsive.

Win32 Dynamic-Link Libraries

Windows NT and Windows 95 support new features for 32-bit DLLs not available to 16-bit DLLs. Win32s supports Win32 DLL features such as:

- Win32 DLLs in Windows 3.1*x* receive the same notifications as on Windows NT and Windows 95 when each process loads or links to the DLL and when each process frees the DLL or terminates. This includes abnormal termination.
- DLLs are initialized in the same order as on Windows NT. This applies to applications with multiple DLLs.
- The Win32s loader searches for Win32 DLLs first in the Win32s directory and then uses the same search algorithm as Windows NT.
- Thread local allocations (**TlsAlloc**) are made system-wide rather than on a per-process basis providing an instance data solution for DLLs on Win32s.

  Another instance data solution is to use **_declspec(***thread***)**. While this method may be more convenient, all of the DLLs that contain the instance data must be loaded implicitly during load time. It is still possible to use **LoadLibrary** to load libraries that don't have static instance data (static data that was declared using **_declspec(***thread***)**). Memory allocation for all of the **_declspec(***thread***)** variables is a 4KB quantum.

File Input and Output (I/O)

The Win32 API provides all functions for manipulating files such as open, seek, move, and rename. The Win32 API is required since calling MS-DOS functions using Interrupt 21H is not supported. Many applications rely on the C run-time library, which in turn relies on Win32 API functions for file I/O. For this reason, the C run-time library is portable and calls Win32s for file I/O.

Floating-Point Emulation

Win32s provides a floating-point trap emulator to support Win32-based applications with in-line math coprocessor instructions. This support allows these applications to function even if they are run on systems without coprocessor hardware. If a coprocessor is present, the trap emulator is not involved.

The same floating-point emulator is used on both Windows NT and Win32s, ensuring compatible results between a Win32-based application running with Windows 3.1*x* and with Windows NT.

Win32-based applications can use structured exception handling (SEH) to handle exceptions generated by the coprocessor. SEH allows the Win32-based application to handle the exception itself and continue or allow the system to handle the exception, generally resulting in termination of the application.

Note Under Win32s, floating-point exceptions are reported as soon as the exception is detected by the processor. On Windows NT, the exception is reported on the following floating-point instruction.

OLE Support

Win32s 1.2 and later provides full support for Microsoft OLE. 32-bit applications that use OLE run without any changes under Win32s.

The OLE2THK library provides the thunking by which 16-bit OLE applications can interoperate with 32-bit OLE applications under Win32s.

Note A 32-bit process cannot call a 16-bit DLL, and vice versa, as mixed-bitness within a process is not supported under Win32 or on Windows 3.1*x*.

Under Win32s, memory that is marshalled between processes must be allocated with **GlobalAlloc**(GMEM_MOVEABLE). Memory allocated as GMEM_FIXED will not be marshalled correctly.

See Shipping Win32s with Win32 Applications for information regarding the OLE files you must ship with your Win32 OLE application for it to run under Win32s.

Printing

Win32 and Win32s support existing printing code. Win32s relies on the existing Windows 3.1*x* printer drivers for output and does not support loading 32-bit Windows NT printer drivers.

Win32s supports all Windows 3.1*x* escapes except those marked as obsolete in the Windows 3.1*x* SDK. Windows NT implements a small set of fundamental escapes for ease in portability, but uses named API functions rather than device-specific escapes to support advanced hard copy output. Therefore, it is critical to query for the support of an escape rather than assume the escape will always be present. If you find that a particular escape is not supported, use an alternate method to accomplish the same task.

Windows 3.1*x* applications can use printer-specific escapes, if available, and should use general-purpose code for doing the same thing if the escape is not supported. This approach should also work in Windows NT. Or, you can call the new Win32 printing API functions that replace the driver-specific escapes when the application runs with Windows NT.

Win32s Programming Details

This section provides detailed programming information that you should be aware of in using the Win32 API and C run-time library in applications that will run on Windows NT, Windows 95, and Windows 3.1. Programming solutions and recommendations are provided to create portable applications and avoid architecture dependencies.

Preemptive Multitasking

Windows NT and Windows 95 are pre-emptive multitasking operating systems; Windows 3.1 relies on message-driven multitasking. This difference can cause compatibility problems when Win32-based applications are run on the different systems unless properly designed and tested.

Data manipulations in DLLs must be guarded by synchronization objects. Windows 3.1 DLL operations are atomic. For example, when an application receives a message, it will be scheduled to handle that message and be given control of the CPU. The application will own the CPU until it checks its message queue. When the application checks its queue, Windows 3.1 may opt to schedule another application. When an application owns the CPU, no other Windows application is scheduled. Therefore, the application handling a message can call a DLL, update data managed in the DLL, and be assured that no other applications call the DLL at the same time.

This is not true of Win32-based applications running on Windows NT or Windows 95. For example, a Win32-based application may call a DLL which begins to update DLL-managed data; the pre-emptive scheduler may pass control of the processor to another process, which could call the same DLL. If the second process also updates data in the DLL, the data will be corrupted. To protect DLL data, the Win32 API provides a set of synchronization objects. These objects are supported in Win32s. Functions like **EnterCriticalSection** are implemented as a NOP and return success even though they are unnecessary in Windows 3.1.

By supporting synchronization calls in Win32s, applications can execute a common code path and simplify programming and testing.

Note It is critical to test an application in both Windows 3.1 and Windows NT to ensure correct operation.

Message Handling

Win32- and Win16-based applications coexist in the system and can pass messages without knowing whether the other application is a 16- or 32-bit application. Win32s maps messages transparently, offering Win32-based applications and 16-bit Windows messages with Win32 semantics.

For private messages (messages defined by an application, not by the Win32 API), Win32s cannot determine the contents of *wParam* and *lParam* message parameters. Therefore, the following effects will occur when passing private messages between Windows applications:

- Private message from Win16 to Win32: *wParam* is 0 extended.
- Private message from Win32 to Win16: high-order word of *wParam* is lost.
- Private message from Win32 to Win32: high-order word of *wParam* is lost.
- Private message from Win16 to Win32 or Win32 to Win32: *lParam* is unaffected.

This behavior has the following impact: Passing a pointer using a private message in *lParam* is safe between two cooperative Win32-based applications; passing a linear 32-bit pointer in *lParam* using a private message to a Win16-based application will result in an invalid pointer when the Win16-based application uses it because Win32s cannot translate this pointer. Because Windows 3.1 is used to deliver messages, the high word of *wParam* is lost even when private messages are being passed only between Win32-based applications. Again, this is a limitation of private messages, not public messages such as WM_COMMAND.

Win32s supports message hooks. Since hooks can be used to intercept messages between applications, Win32s assures that hooks installed by Win16-based applications can intercept messages destined for Win32-based applications; and similarly Win32 hooks can intercept messages destined for Win16-based applications. In each case the 16- or 32-bit message hook will intercept messages for Win16- and Win32-based applications, but translated into the appropriate 16- or 32-bit message form.

Dynamic Data Exchange

Win32s supports Dynamic Data Exchange (DDE). However, while it is possible to use DDE on Windows NT or Windows 95 without using the DDE message-packing functions, using DDE under Win32s requires that these functions are used. The message-packing functions are **PackDDElParam**, **UnpackDDElParam**, **ReuseDDElParam**, and **FreeDDElParam**.

Memory Management

Windows 3.1 uses a single address space in which all Windows (16- and 32-bit) applications run. Therefore, DLL data segments are shared among all processes that call the DLL. Windows NT has separate address spaces with 2GB of virtual memory available for each process. The default DLL behavior is to have private DLL data, also known as instance data. Windows NT DLLs can be built to have shared DLL data for compatibility with Windows applications that require this feature.

In order to specify that DLL data should be shared, add the following lines to your Win32 DLL's DEF file:

```
SECTION
.bss READ-WRITE-SHARED    ; Share uninitialized global variables
.data READ-WRITE-SHARED   ; Share initialized global variables
```

Note: If your port of a 16-bit Windows-based application to Win32 relies on shared DLL data, you should build the DLL indicating shared data. The default linker behavior is to create instance data DLLs resulting in shared DLL data on Windows 3.1 and instance data on Windows NT.

Note: The Win32s loader loads the DLL data section as SHARED, regardless of the section attribute.

To facilitate communication and passing of data between Win32-based applications and other Win16-based applications (and Windows 3.1 itself), Win32s supports the various memory allocation functions differently than on Windows NT or Windows 95.

On Windows NT and Windows 95:

- **VirtualAlloc** supports sparse memory allocation.
- **GlobalAlloc** maps to **HeapAlloc** (always commits memory).
- **LocalAlloc** maps to **HeapAlloc**.
- **HeapAlloc** implements suballocation from large memory block allocated using **VirtualAlloc**.
- C run-time **malloc** maps to **HeapAlloc**.

On Win32s:

- **VirtualAlloc** supports sparse memory allocation.
- **GlobalAlloc** relies on Windows 3.1 **GlobalAlloc** and always allocates committed and fixed memory.
- **LocalAlloc** maps to **HeapAlloc**.
- **HeapAlloc** implements suballocation from large memory blocks allocated using **VirtualAlloc**.
- C run-time **malloc** maps to **HeapAlloc**.

The primary difference is the behavior of **GlobalAlloc**. Win32s relies on Windows 3.1 for global allocation. This allows a 32-bit application to simply pass the handle to global memory to Windows (such as the clipboard) or to another 16-bit application. The memory can then be locked, unlocked, resized or freed; the 16-bit side (system or application) is never aware that the memory was allocated by a 32-bit process. This is not true for the other memory allocation functions. In these cases Win32s has to create and manage handles dynamically using a private handle cache.

The **VirtualAlloc** and **HeapAlloc** functions are better memory allocation methods for application-manipulated data. The **GlobalAlloc** function is more efficient for data that will be passed back and forth between the Win32-based application and system or other applications.

Allocating global memory with the GMEM_FIXED flag locks the pages. In most cases, this is undesirable and affects the overall performance of the system. It is recommended that you allocate memory as moveable and obtain a pointer to the memory using **GlobalLock**.

Thread Local Storage is supported on Win32s even though threads are not. TLS is useful in Win32

DLLs on Win32s to create DLL instance data. By default, all global variables in a DLL on Windows 3.1*x* are shared by all processes that call the DLL. This can be inconvenient when the DLL must maintain global information that applies only to a particular process calling the DLL. The TLS implementation in Win32s creates a system-wide index such that a unique TLS index is available for each process that links to a DLL.

In the following DLL initialization code fragment, the DLL will allocate a TLS index for each process that attaches to the DLL on Windows NT and Windows 95. On Win32s, a TLS index is only allocated when the first process attaches to the DLL. Each Win32 process can use **dwIndex** in calls to **TlsSetValue** and **TlsGetValue** to store per-process data (such as a pointer to allocated data unique to each process).

Another way to look at TLS support on Win32s is that each process in Windows 3.1 uses the TLS index like a thread does on Windows NT or Windows 95.

The following DLL initialization code fragment illustrates how to initialize TLS in a DLL that will work on Windows NT, Windows 95, and Win32s:

```
BOOL APIENTRY DllEntryPoint(HINSTANCE hinstDll, DWORD fdwReason,
        LPVOID lpvReserved)
{
    static BOOL fFirstProcess = TRUE;
    DWORD  dwVer = GetVersion();
    BOOL   fWin32s = (dwVer & 0x80000000) &&
                     (LOBYTE(dwVer)<4);
   static DWORD  dwIndex;

    if (dwReason == DLL_PROCESS_ATTACH) {
        if (fFirstProcess || !fWin32s) {
            dwIndex = TlsAlloc();
         }
         fFirstProcess = FALSE;
    }
    .
    .
    .
}
```

There should be a matching platform test in the DLL **TlsFree** cleanup code.

C Run-time Support

Win32-based applications that rely on C run-time routines may be linked statically to the application or dynamically to CRTDLL.DLL, a C run-time DLL. CRTDLL.DLL is part of the Win32s system and should be redistributed. Therefore, an application can rely on the existence of the C run-time DLL on Windows NT, Windows 95, and Win32s.

Using CRTDLL.DLL is useful for applications that have an executable and multiple DLLs that all make C run-time calls. The CRTDLL.DLL is also useful for applications that consist of multiple executables that share a single or set of application DLLs which all rely on C run-time calls.

Because of the shared system memory of Windows 3.1, some aspects of CRTDLL.DLL cannot be supported. Multiple applications can link and use CRTDLL.DLL simultaneously, but global variables such as **__argc_dll** and **__argv_dll** are not exported by the Win32s version of CRTDLL.DLL. The portable method for determining the command-line parameters would be to call **GetCommandLine**. There are several other similarly affected global variables, such as **_environ**.

The portable method for getting environment variables when using CRTDLL.DLL is by calling Win32 APIs **GetEnvironmentStrings** or **GetEnvironmentVariable** instead of **_environ**.

It is not recommended to statically link to a CRT library. Win32s applications should link to CRTDLL.DLL or the CRT library provided by your compiler vendor--for example, MSVCRT20.DLL.

Note that your C compiler vendor may have its own form of C run-time in a redistributable DLL.

File Input and Output

In network environments especially, or on a single system with multiple applications opening a single file, applications should indicate how they would like to share, or not share, files explicitly. The flag semantics for Win32s and Windows NT are the same for **OpenFile**. For **CreateFile**, however, there are differences. If **fdwShareMode** is a zero, Win32s interprets it as compatibility mode (OF_SHARE_COMPAT), while Windows NT interprets it as deny all (OF_SHARE_DENY_ALL). Under Win32s, **CreateFile** and **fopen** do not support OF_SHARE_DENY_ALL. Windows NT does not support compatibility mode (OF_SHARE_COMPAT). For further information, see the **OpenFile** reference topic.

It is very important to enable file-sharing and locking. When running on Windows 3.1*x*, be sure that SHARE.EXE has been loaded before starting Windows. Windows for Workgroups provides file-sharing and locking support by default, so you should not load SHARE.EXE.

International Support

Win32-based applications that are designed for international markets will find localized versions of Win32s and code page support for manipulating resources.

Localized Versions of Win32s

Since Win32s consists primarily of a mapping layer, it is language independent. Therefore, Win32s can be installed on localized versions of Windows 3.1 or Windows for Workgroups. Win32s does have several low-level error messages that are displayed when memory is low, the Win32s loader encounters a corrupt binary, and so on. These error message strings are isolated to the VxD and one 16-bit DLL, W32S.386 and W32SYS.DLL, respectively. Additionally, system messages are provided by NTMSG.DLL.

To ship a localized version of a Win32-based application with Win32s, you must replace the English versions of W32S.386, W32SYS.DLL, and NTMSG.DLL with the appropriate language version of these files from the \MSTOOLS\WIN32S\NLS directory according to the following table:

```
w32s.deu, w32sys.deu:    German
w32s.esp, w32sys.esp:    Spanish
w32s.fra, w32sys.fra:    French
w32s.ita, w32sys.ita:    Italian
w32s.sve, w32sys.sve:    Swedish
```

Localized versions of NTMSG.DLL are generated with **mkntsmg,** which can be found in \MSTOOLS\WIN32S\BIN. The syntax for **mkntsmg** is:

mkntmsg [**-v**] [**-h**] [**-o** *outfile*] *infile*

-v
(Verbose) Displays the message groups.

-h
Displays the usage message.

-o *outfile*
Specifes the output file name.

infile
Should be KERNEL32.DLL taken from a localized version of Windows NT.

This current set of languages matches those supported in the first release of Windows NT. If your application requires additional language support, contact Microsoft Product Support Services, which are most easily reached on CompuServe. For CompuServe information, see the *Win32 SDK Getting Started* documentation.

Code Page and Unicode Translation Support

Windows 3.1 does not support Unicode, so Win32s 1.25 has been implemented without Unicode support. Win32 application resources, however, such as dialog boxes, menus, and message tables consist of Unicode strings. Win32s translates these resources automatically using the Windows 3.1 code page. However, some applications may wish to control string translation using other code pages.

Win32s version 1.25 supports Code Page/Unicode translation using the following functions:

```
MultiByteToWideChar
WideCharToMultiByte
IsValidCodePage
GetCPInfo
IsDBCSLeadByte
```

These functions can be useful when creating dialog boxes dynamically within an application. Dialog box templates require Unicode, not ANSI, character strings.

Win32s 1.25 installs the following OEM and ANSI code pages by default:

CTYPE.NLS
LOCALE.NLS
L_INTL.NLS
P_037.NLS
P_10000.NLS
P_10001.NLS
P_10006.NLS
P_10007.NLS
P_10029.NLS
P_10081.NLS
P_1026.NLS
P_1250.NLS
P_1251.NLS
P_1252.NLS
P_1253.NLS
P_1254.NLS
P_437.NLS
P_500.NLS
P_737.NLS
P_850.NLS
P_852.NLS
P_855.NLS
P_857.NLS
P_860.NLS
P_861.NLS
P_863.NLS
P_865.NLS
P_866.NLS
P_869.NLS
P_875.NLS
P_932.NLS
P_936.NLS
P_949.NLS
P_950.NLS
SORTKEY.NLS
SORTTBLS.NLS
UNICODE.NLS

The table below lists the code pages supported by these .NLS files:

The following table shows various Code Pages and the support associated with each code page. It also lists whether or not a code page is a valid ACP or OEMCP.

CP ID – the Code Page ID
Name – the canonical name of the character set
ACP –eligibility for ACP
OEMCP– eligibility for OEMCP

Code Page Identification Numbers

CP	Name	ACP	OEMCP	Windows NT 3.1	Windows 95	Windows NT 3.5	"Cairo"

ID						
1002000	Unicode (BMP of ISO 10646)		x		x	x
1252050	Windows 3.1 Eastern European	x	x	x	x	x
1252051	Windows 3.1 Cyrillic	x	x	x	x	x
1252052	Windows 3.1 US (ANSI)	x	x	x	x	x
1252053	Windows 3.1 Greek	x	x	x	x	x
1252054	Windows 3.1Turkish	x	x	x	x	x
1252055	Hebrew	x		x	x	x
1252056	Arabic	x		x	x	x
1252057	Baltic	x		x	x	x
437	MS-DOS United States		x	x	x	x	x
708	Arabic (ASMO 708)		x		x	x	x
709	Arabic (ASMO 9449+, BCON V4)		x		x	x	x
710	Arabic (Transparent		x		x	x	x

Arabic)							
772 0	Arabic (Transparent ASMO)		x		x	x	x
773 7	Greek (formerly 437G)		x	x	x	x	x
775	Baltic		x		x	x	x
850	MS-DOS Multilingual (Latin I)		x	x	x	x	x
852	MS-DOS Slavic (Latin II)		x	x	x	x	x
855	IBM Cyrillic		x	x	x	x	x
857	IBM Turkish		x	x	x	x	x
860	MS-DOS Portuguese		x	x	x	x	x
861	MS-DOS Icelandic		x	x	x	x	x
862	Hebrew		x		x	x	x
863	MS-DOS Canadian-French		x	x	x	x	x
864	Arabic		x		x	x	x
865	MS-DOS Nordic		x	x	x	x	x
866	MS-DOS Russian (USSR)		x	x	x	x	x
869	IBM Modern Greek		x	x	x	x	x
874	Thai	x	x		x	x	x
932	Japan	x	x		x	x	x

2							
93	Chinese (PRC, Singapore)	x	x		x	x	x
94	Korean	x	x		x	x	x
95	Chinese (Taiwan, Hong Kong)	x	x		x	x	x
87	Thai		x		x	x	x
13	Korean (Johab)						
10	Macintosh Roman			x	x	x	x
10	Macintosh Japanese				x	x	x
10	Macintosh Greek I			x	x	x	x
10	Macintosh Cyrillic			x	x	x	x
10	Macintosh Latin 2			x	x	x	x
10	Macintosh Icelandic				x	x	x
10	Macintosh Turkish			x	x	x	x

037	EBCDIC	x	x	x
500	EBCDIC "500V1"	x	x	x
10000	EBCDIC	x	x	x
10029	EBCDIC	x	x	x
10079	EBCDIC	x	x	x

Win32s 1.25 installs all of these OEM code pages by default.

Note Use of code pages other than UNICODE.NLS and P_1252.NLS requires several file handles and 150K - 220K of virtual memory. This extra resource will only be consumed when one of the five code page APIs listed above is called.

The code pages installed by default enable multi-locale systems. These code pages are provided in \MSTOOLS\WIN32S.

The OEM code page that is installed should match the code page used by MS-DOS. This has ID 437 in the U.S. and, generally, ID 850 for Western Europe. Win32s 1.2 installs both of these code pages by default. Your Setup program will need to install other OEM code pages to match MS-DOS code pages for other languages.

See Shipping Win32s with Win32 Applications for installation instructions.

Performance Considerations

Calls to the Win32 API are translated before they are passed to Windows. This involves translating pointers and repacking parameters on the stack, among other operations. Therefore, there is an overhead associated with calling the Windows API. For functions that require a large amount of processor time to complete (such as **BitBlt**), the thunking overhead as a percentage of total API execution time is less than one percent. Other functions—for example, functions that simply query status—have a relatively high thunking overhead. Test scenarios that call a broad selection of Win32 functions take approximately ten percent longer due to the thunking layer.

Therefore, an application that calls only Win32 functions will experience a 10 percent degradation. However, real applications have a mix of system calls and time spent in the 32-bit routines of the application, and generally will see a performance increase.

Win32s is a perfect candidate for applications that are data- and memory-intensive or calculation-intensive, such as CAD packages, desktop publishing packages, image manipulation tools, spreadsheet programs, and simulation software. Manipulating data and performing calculations in 32-bit mode improves the performance of these applications significantly. Win32-based applications should see performance improvements despite the slight overhead resulting from 32-bit to 16-bit translation (thunking) that occurs during Windows API calls.

Several Win32-based applications have been used to do performance analysis of Win32s by comparing performance between 16- and 32-bit versions of the same applications. For GDI-intensive operations used in these applications, the performance was found to be roughly the same. For numeric computation and for traversing data, the Win32s versions of the applications were up to 2 times faster.

The overall effect is that most Win32-based applications using Win32s will see a performance gain over their 16-bit Windows counterparts, with the added benefit that these new applications are native 32-bit applications on Windows NT.

Using the following techniques can significantly improve the performance of your Win32-based application on both Windows 3.1 and Windows NT:

- Use the Working Set Tuner provided with the Win32 SDK.
- Use the **PolyLine** function rather than **MoveTo**/**LineTo** operations.
- Use **PolyPolygon** instead of multiple calls to **Polygon**.
- Use a reference between the Win32-based application and the system to pass local variables (stack variables).

Using **PolyLine** saves time in Win32s by minimizing the number of thunks that must pass through from the Win32-based application to Windows 3.1*x* system code. This is beneficial on both Win32s and Windows NT.

Pointer translation from 32 bits to 16 bits can be a time-consuming operation for Win32s. Different types of memory (**GlobalAlloc**, **LocalAlloc**, **VirtualAlloc**, global data, local data) take different amounts of time to translate pointers. In general, pointers to local variables will always be translated fastest since they are on the application's stack, and Win32s has optimized translation of the stack pointer.

Advanced Features

Because the entire Win32 API is exported by Win32s, any Win32-based application can be loaded into Windows 3.1*x*. All functions will be fixed-up correctly by the loader. Applications that call Win32 functions that cannot be implemented in Windows 3.1, such as paths, threads, transformations, or asynchronous file I/O, will find that these functions fail and return errors.

Unsupported APIs return error codes appropriate for the function called. Win32s will set the last-error code to ERROR_CALL_NOT_IMPLEMENTED, which can be retrieved by **GetLastError**. However, Win32-based applications do not need to rely exclusively on error return codes, but can determine which Windows platform they are running on (Windows 3.1 or Windows NT) by calling the **GetVersion** function. The high-order bit of the **DWORD** return value is *zero* when the Win32-based application is running on Windows NT and is *one* if the application is running on Windows 3.1 or Windows 95.

An application can selectively, and as a run-time feature, implement different functionality when the application is run with Windows 3.1 using Win32s or when it is run with Windows NT. For example, a graphics application could offer Bezier-curve drawing tools when the paint program is run on Windows NT. The same tool could emulate this functionality itself or not offer the feature with Windows 3.1*x*. The application can use the error return from these unsupported Win32 functions in Win32s to determine the proper plan of action.

When running on Windows NT, an application can spawn background threads to complete tasks, something it cannot do when running on Windows 3.1*x*, where it runs only as a single-threaded application. But, consider that all Win32-based applications automatically benefit from the scheduler on Windows NT even if they do not use multiple threads. Win32-based applications execute with an asynchronous messaging model on Windows NT, which allows users to switch away from applications that are tied up in long calculations. Taking advantage of threads on Windows NT is not required, and most applications will rely on the asynchronous messaging model advantage.

Mixing 16-bit and 32-bit Code

Windows NT does not support mixing 16- and 32-bit code. Such support is contrary to the portable design of Windows NT running on *x*86 (CISC) and R4000 (RISC) systems. The Win32 SDK tools generate only 32-bit code, and the Windows NT linker and loader have no support for fixing up 16-bit segmented addresses.

Therefore, for binary compatibility of a Win32-based application running on Win32s and Windows NT, mixing is not supported by Win32s. Mixing is not limited to embedding 16-bit routines in a Win32-based application, such as static library code. There is no support for Win32-based applications directly loading 16-bit DLLs or vice versa.

Applications that rely on 16-bit DLLs, such as licensed DLLs with difficult-to-obtain 32-bit versions, have several choices:

- Use a client/server architecture where all 16-bit DLLs are bound to a small, custom 16-bit server application. This custom 16-bit server can communicate with the Win32-based application using DDE, shared memory, or other IPC mechanism. The client/server solution will work in Win32s and Windows NT.
- A unique solution provided by Win32s allows Win32-based applications to use existing 16-bit DLLs and device drivers. The Universal Thunk, discussed later in this document, is not supported on Windows NT but is available as a stepping stone for moving applications to 32-bit on Windows 3.1*x*.

Device Drivers

Windows NT runs applications in user mode, but only kernel-mode device drivers running in privileged mode can access devices. Because of fundamental differences in architecture between Windows 3.1 and Windows NT, Win32s does not support the Windows NT device-driver model. For binary compatibility with Windows NT, Win32-based applications should not access hardware directly within the application.

This poses a problem similar to mixing 16-bit and 32-bit code. The general solution is to build your Win32-based application such that the main application has a standard interface to a Win32 DLL (service provider). The Win32 DLL uses a client/server design to request/transfer data to a 16-bit server responsible for hardware access on Windows 3.1*x*.

To support this application in Windows NT, the DLL should be replaced by a Windows NT-specific DLL that uses IOCTLs to communicate directly with the kernel-mode driver, or uses Win32 APIs for supported devices. This solution isolates the platform-specific code to a DLL, which can be correctly installed during setup.

This mixing restriction creates an interesting scenario for printing. The 16-bit Windows applications load and call printer drivers directly, by using calls such as **LoadLibrary**, **GetProcAddress**, and **ExtDeviceMode**. Win32s handles these printer drivers specifically by creating a mapping thunk dynamically. The address returned to the Win32-based application by **GetProcAddress** is actually the address to the thunk that makes the 32/16 transition and calls the printer driver. This Win32s solution for printing has been generalized for Win32-based applications with the *universal thunk* solution.

Universal Thunk

Binary compatibility and the requirement to provide natural migration to Windows NT and Windows 95 creates a portability problem for developers who have Windows applications that require device drivers. Windows NT does not support mixing 16- and 32-bit code in a Win32 process. The mixing restriction also prevents Win32-based applications from calling 16-bit DLLs.

A number of IPC mechanisms, such as DDE, allow data to be passed between Win32 and 16-bit Windows processes, but the bandwidth is not sufficiently high for some types of applications. The universal thunk (UT) allows a Win32-based application to call a routine in a 16-bit DLL. There is also support for a 16-bit routine to call back to a 32-bit function. The simple Win32s thunk used to implement this design also translates a data pointer to shared memory, from flat to segmented form, allowing large amounts of memory to be transferred between drivers and Win32-based applications.

This design allows a Win32-based application to isolate driver-specific routines in a 16-bit DLL. The Win32-based application remains portable across Windows 3.1*x*, Windows 95, and Windows NT; in Windows NT, the 16-bit DLL is replaced with a 32-bit DLL that communicates to devices by using the Windows NT model. A similar DLL would be written for Windows 95.

Universal Thunk Design

The following figure illustrates a Win32-based application using a Win32 service DLL on Windows NT or Windows 95.

{ewc msdncd, EWGraphic, group10903 0 /a "SDKS-1.bmp"}

The next figure illustrates how the same Win32-based application running on Windows 3.1*x* uses the UT mechanism to access services provided by a Win16 DLL. The figure illustrates thunking from 32-bit to 16-bit; thunking in the reverse direction works in a similar way.

The same version of the 32-bit application runs on Windows 3.1*x,* Windows 95, and Windows NT. For the service provider, different modules are required. They all provide the very same services, one using Win32 DLL on Windows NT, and another using 16-bit DLL by way of UT on Win32s.

{ewc msdncd, EWGraphic, group10903 1 /a "SDKS-2.bmp"}

UTRegister registers a UT with Win32s that can be used to access 16-bit code from a 32-bit application running on Win32s. **UTRegister** will automatically load the 16-bit DLL specified by name as a parameter to this function. The 16-bit DLL exports two functions: an initialization function **InitFunc** and function **UTFunc. InitFunc** is called once when **UTRegister** is called and is optional. **UTFunc** is called indirectly using a 32-bit callable stub from the Win32-based application.

The thunk can be destroyed by calling **UTUnRegister**.

Registering the UT enables two options for communicating between 32-bit and 16-bit routines. The first option is to allow a Win32-based application to call a 16-bit routine, passing data using global memory. This is a Win32 application-initiated data transfer mechanism. The second option is to register a callback routine by which 16-bit code can callback into a 32-bit routine in a Win32 DLL. Again, global memory is used to transfer data.

UTRegister will result in Win32s loading the specified DLL (using **LoadLibrary**) with normal Windows 3.1*x* DLL initialization occurring. Win32s will then call the **InitFunc** routine (indirectly using the UT), passing this function a 16:16 alias of the 32-bit callback function. The **InitFunc** routine can return data in a global memory buffer. **UTRegister** will also create a 32-bit callable stub for **UTFunc** and return its address to the 32-bit DLL. Also, a 16-bit callable stub might be created to reflect the 32-bit callback and will be passed to the 16-bit DLL using its **InitFunc.**

The UT will translate the 32-bit linear pointer to shared memory to a 16:16 segmented pointer for the 16-bit **UTFunc** routine. The application must define the format and packing of the data in the global memory. Care should be taken when passing information in structures using global memory since some data types, such as **int**, are 32-bits in Win32 and 16-bits in Windows 3.1*x*.

The UT stub and the callback thunk are associated with the 32-bit module whose module handle is passed to **UTRegister**. Only one UT can be registered per 32-bit DLL at any given time.

UTUnRegister allows the dynamically created UT to be destroyed and the 16-bit DLL freed. Win32s will clean up these resources automatically when the Win32 process terminates, either normally or abnormally.

1. Win32s will destroy the UT and call **FreeLibrary** for the 16-bit DLL when:
   - **UTUnRegister** is called.
   - The module that created the UT is unloaded, either normally or abnormally.
2. The shared memory accessed using the thunked data pointer will be freed as part of normal process cleanup if the process itself does not free the memory.
3. The 32-bit callback function should not be used by 16-bit interrupt handlers. The Win32 API does not support locking memory pages; therefore, an interrupt service routine that calls back into 32-bit code cannot guarantee that the code is currently paged into memory.

Translation List

One of the parameters passed to the callable stub (either 32-bit or 16-bit) is a translation list. If you want to pass a reference to a structure containing pointers, you should use a translation list (see the following figure), which indirectly points to all the pointers to be translated during the thunking process.

{ewc msdncd, EWGraphic, group10903 2 /a "SDKS-3.bmp"}

Upon a call to UT callable stub:

```
(*pfnMyDispatch)(lpPerson, SERVICE_ID, pTranslationList);
```

the thunking process will translate the *lpPerson* pointer, scan the *lpTranslationList* and translate in place all the pointers, change the stack, and call the relevant function.

Universal Thunk Implementation Notes

The following information discusses implementation issues that affect application performance. The UT functions, parameters, and structures are defined in the Win32s API Reference. The UT headers are located in the \MSTOOLS\WIN32S\UT directory.

1. Include the file W32SUT.H when using UT services. On the 32-bit side, define W32SUT_32 before including the file. On the 16-bit side, define W32SUT_16.
2. Link a 32-bit DLL using UT services with the library W32SUT32.LIB. A 16-bit DLL should be linked with the library W32SUT16.LIB.
3. **UTRegister** and **UTUnRegister** can be loaded by calling **GetProcAddress** on KERNEL32.DLL, or by linking to W32SUT.LIB. **UTLinearToSelectorOffset** and **UTSelectorOffsetToLinear** are exported by WIN32S16.DLL
4. Only one UT may exist for each Win32 DLL. Additional calls to **UTRegister** will fail until **UTUnRegister** is called for that DLL.
5. Win32s will destroy the UT and call **FreeLibrary** for the 16-bit DLL when:
   - **UTUnRegister** is called
   - The Win32 DLL is freed, either normally or abnormally
6. Memory allocated by 16-bit routines using **GlobalAlloc** should be fixed using **GlobalFix**. It must be translated to flat address before it can be used by 32-bit code. Translation will be performed by Win32s if passed as **lpBuff** or by using the translation list. It can also be translated explicitly using **UTSelectorOffsetToLinear**.
7. The 32-bit callback function should not be used by 16-bit interrupt handlers. The Win32 API does not support locking memory pages, therefore an interrupt service routine that calls back into 32-bit code cannot guarantee that the code is currently paged into memory.
8. Exception handling can be done in 32-bit code only. Exceptions in 16-bit code will be handled by the last 32-bit exception frame.
9. While it is possible to call into 32-bit code from a 16-bit task using Universal Thunks, that 16-bit task will not have a proper 32-bit context, and many things (such as TLS indices) will not work.

Two samples are provided on the Win32 SDK CD-ROM that illustrate how to use and build a UT application:

- \MSTOOLS\WIN32S\UT\SAMPLES\UT_DEF illustrates the fundamental pieces required to use the UT.
- \MSTOOLS\WIN32S\UT\SAMPLES\UTSAMPLE is a buildable UT sample which calls several 16-bit functions and returns data to the Win32-based application.

Note The UT has been tailored to Windows 3.1*x*, which provides a single address space for all processes. Applications that use the UT may have dependencies to Windows 3.1. The UT should be considered a stepping stone to full 32-bit applications. It is very important to isolate UT code in separate DLLs. These DLLs can be more easily replaced to run on Windows NT or Windows 95. The DLL can be replaced with platform-specific versions, isolating the application.

Win32s Development with the Win32 SDK

The Win32 SDK combined with a Win32 compiler provides all the tools necessary for creating a Win32-based application on Windows NT or Windows 95.

After a Win32-based application has been debugged and fully tested on Windows NT and Windows 95, you are ready to install Win32s and your Win32-based application on Windows 3.1*x* for final testing and verification. The Win32 SDK provides floppy-disk images containing a redistributable version of Win32s with a Setup program and FreeCell, a card game test application. You can choose to bundle these disks with your product or use the Win32s Setup Toolkit development files to customize your own application and Win32s installation program. For further details on Win32s redistributable files, refer to the Win32 SDK License Agreement.

Installing Win32s Development Files

The Win32s system files, Setup Toolkit for Win32s, and the development files are provided on the Win32 SDK CD-ROM in the \MSTOOLS\WIN32S directory. These files are intended for use on a Windows 3.1*x* system. Two versions of Win32s are provided to aid debugging:

- A debug version with additional Win32s asserts and diagnostic messages.
- The nondebug (end-user) version of Win32s.

Symbol files for both debug and nondebug systems are provided to obtain symbolic information in stack dumps and when using the kernel debugger.

Also included are the sources to the Setup program used in the Win32s Setup program. These files should be used in conjunction with the 16-bit Microsoft Setup Toolkit to create and customize your own setup program.

To install the Win32s development files onto a Windows 3.1*x* system, first install Win32s by using the Win32s Setup program described in the Introduction to Microsoft Win32s, then follow these steps:

1. Start MS-DOS with appropriate software for accessing a local CD-ROM drive or a remote CD-ROM drive over the network.
2. Access the Win32 SDK CD-ROM.

   Type the drive letter and press enter. (For example—**x**:, where x: is the CD-ROM drive or network drive containing the Win32 SDK CD-ROM).
3. Change directories to the Win32s files by typing **cd \mstools\win32s**.
4. Type **install** and press ENTER.
5. The INSTALL batch script will provide usage information on how to specify the destination directory for the Win32s files.

Porting Tool

Win32s allows applications to call any Win32 function. Win32 functions that cannot be supported in Windows 3.1*x* generally return errors. A spreadsheet listing all Win32 APIs and whether the API is supported on Win32s, Windows 95, or Windows NT is located on the Win32 SDK CD-ROM in \MSTOOLS\LIB\I386\WIN32API.CSV, a comma-separated file.

To help programmers determine whether they have inadvertently referenced unsupported Win32 functions, the Win32 SDK provides a porting tool and a data file, WIN32S.DAT, that lists all unsupported Win32 functions on Win32s. Read WIN32S.DAT into the porting tool, load your application source code, and let the porting tool scan your sources.

To read the file into the porting tool:

1. First install the Win32 SDK.
2. Rename PORT.INI in \MSTOOLS\BIN to PORT.DAT.
3. Rename WIN32S.DAT in \MSTOOLS\BIN to PORT.INI.
4. Start Porting Tool (from the Win32 SDK Program Group).
5. Load a source file to be analyzed.
6. Select Interactive or Background port to have unsupported Win32 functions found and highlighted.

Note For more information on using the porting tool, run PORTTOOL and bring up Help.

Debugging and Testing

The Win32 SDK tools running on Windows NT or Windows 95 provide the primary development and debugging environment for creating Win32-based applications. A Win32-based application should be fully functional and debugged using WinDbg (or equivalent debugger) on Windows NT and Windows 95 before running the application on Win32s. There are several debugging solutions available to help test and debug Win32-based applications running on Windows 3.1*x*: remote WinDbg, debugging DLLs, profiling DLLs, and the Windows 3.1*x* kernel debugger. Other debugging tools are available from tools vendors such as Borland, NuMega, Symantec, Microsoft, and others.

Remote WinDbg

The remote WinDbg solution makes it possible to debug your Win32-based application running on Windows 3.1*x* (a second system) from your Windows NT or Windows 95 development machine (primary system). The Windows NT (or Windows 95) system is the development system with full sources for the application, where WinDbg debugs the Win32 binary running on the secondary Windows 3.1*x* system. Since primary development and debugging should already be complete on Windows NT or Windows 95, this remote debugging technique should only be necessary to debug unique problems that occur when the Win32-based application runs on Windows 3.1*x*.

{ewc msdncd, EWGraphic, group10904 0 /a "SDKS-1.bmp"}

{ewc msdncd, EWGraphic, group10904 1 /a "SDKS-2.bmp"}

Binary debugging data is passed between the Windows 3.1*x* and Windows NT or Windows 95 systems, so a cable that supports hardware handshaking (HW) is recommended. XON/XOFF flow control is also supported if a minimally wired NULL modem cable is used, but transfer is somewhat slower since data must be translated for transfer. The preceding figure illustrates standard cable wiring that should be used with the remote debugger.

Note The cable layouts in the preceding figure are for standard DB-9 serial connectors. Be sure to refer to your serial card manufacturer's manual for complete cabling information.

When setting up the remote debugger transport, be sure to specify XON/XOFF or HW (hardware handshaking) to match the cable that you are using. Also, make sure that you match the handshaking of the WinDbg and Remote WinDbg transports; both should be XON/XOFF or both should be set up for HW.

Remote WinDbg has better response with high communication baud rates and using HW rather than XON/XOFF control. Typically, you should use a baud rate of 19.2K. Test your connections before attempting to debug by using the Terminal program that ships with both Windows 3.1*x* and Windows NT or Windows 95.

Install Win32s and your application on the Windows 3.1*x* system. The following remote debug components should also be installed on the Windows 3.1*x* system into a directory that is referenced by the PATH environment variable:

- WDBG32S.EXE
- DM32S.DLL
- TLSER32S.DLL

Note WinDbgRm from Windows NT 3.5 no longer runs on Win32s. Use WDBG32S instead.

These files are located on the Win32 SDK CD-ROM in the \MSTOOLS\BIN\I386 directory. The following figures illustrate how to setup the WinDbg and WDBG32S transports to use COM1 at 19,200 baud with XON/XOFF control.

{ewc msdncd, EWGraphic, group10904 2 /a "SDKS-3.bmp"}

{ewc msdncd, EWGraphic, group10904 3 /a "SDKS-4.bmp"}

WinDbg minimizes the amount of information transmitted across the serial line by accessing symbols out of the local version of the application binaries located on the development system. WinDbg is designed to access the source code on the development system, saving enormous communication overhead that would be required if symbols and source code information were transferred over the serial line.

The remote debugging environment requires that binaries be located on the same drive/directory on both the development and target systems. For example, if WIN32APP.EXE is built from sources in a C:\

DEV\WIN32APP directory, the binary should be located in this path on both systems. If you build your source files by specifying fully qualified paths for the compiler, the compiler will place this information with the debug records which will allow WinDbg to automatically locate the appropriate source files.

WinDbg expects the sources to be in the same directory where the binary is built, but WinDbg allows browsing for them if the sources are not found in the default location.

The Win32 SDK allows applications to be built with CodeView™, COFF debugging symbols, or both. The makefiles provided with the Win32 SDK samples illustrate the compiler and linker switches required to build CodeView symbols for WinDbg. For compiling, use **-Od** and **-Zi** to turn off optimization and specify CodeView symbols, respectively. Specify **-debug:full** and **-debugtype:cv** when linking with LINK32. Makefile macros (such as cdebug and ldebug) in WIN32.MAK can be used to make this easier.

Debugging Session Example

To initiate a debugging session, begin with the host (Windows NT or Windows 95) system:

- **windbg** C:\DEV\WIN32APP\WIN32APP.EXE (where WIN32APP.EXE is the name of your binary and C:\DEV\WIN32APP is the location of the binary on both systems).
- Select the WinDbg menu Options/Debugger DLLs and set the Transport to SERIAL192. Set the serial port communication parameters.

On the Windows 3.1*x* system:

- Run WDBG32S.EXE.
- Select the WDBG32S menu Options\TransportDLLs and set the Transport to SERIAL192. Adjust the serial port parameters to match the WinDbg parameters on the Windows NT system.
- Choose Connect.

Now return to the host system:

- Step into the application. This will establish the communication connection.
- Once a connection is established, WDBG32S will disappear. When the debugging session ends or the connection is broken, WDBG32S will reappear to allow you to reconnect or exit.

At this point you can now debug the application just as you would if you were debugging locally on the host system; you can set breakpoints, single step, display locals, structures, and so on.

Note WinDbg is designed for 32-bit and 16-bit debugging on Windows NT, but only 32-bit debugging on Win32s. Therefore, do not use WinDbg to trace into universal thunks or attempt to set breakpoints in 16-bit code. Using both the kernel debugger (WDEB386) and WinDbg simultaneously is not recommended.

Win32s Debugging DLLs

The Windows 3.1*x* SDK provides two environments for debugging or testing your Windows applications: a debugging version of the retail Windows product and a nondebugging version. The same is true for Win32s shipped with the Win32 SDK; there are debug and nondebug versions of Win32s.

The debugging version of Windows 3.1*x* consists of a set of DLLs that replace the Windows system DLLs of the retail product. The debugging DLLs provide error checking and diagnostics messages that help you debug a Windows application. Along with the DLLs, symbol files are also provided to help trace calls into Windows and Win32s.

During application development, you should use the debugging version of Windows. Switch to the nondebug versions for final application testing. To simplify switching between debug and nondebug Win32s binaries, use the SWITCH.BAT file located in the \MSTOOLS\WIN32S\BIN directory. Usage information is provided by invoking this batch file with no parameters.

Note Do not ship the debugging version of the Win32s files with your application. Not only would the debug versions undermine your application's performance, the license agreement on Win32s Redistributables only covers the nondebug files.

The Win32 SDK provides debugging versions of the Win32s DLLs. It is necessary to obtain a Windows 3.1*x* SDK for the Windows 3.1*x* debugging DLLs. The Windows 3.1*x* SDK also provides additional details on how to use the debug DLLs.

Automated Testing Using Microsoft Test

Microsoft Test is an automated testing tool that facilitates building test scripts for regression testing, profiling, and quality assurance.

The same scripts built with 32-bit Microsoft Test on Windows NT or Windows 95 can be used with the 16-bit version of Microsoft Test for Windows 3.1*x* to test Win32-based applications using Win32s. This will help verify and guarantee that your Win32-based application functionality is identical on Windows NT, Windows 95, and Windows 3.1*x*.

The Microsoft Test product is available separately from Microsoft and is not part of the Win32 SDK that ships 32-bit tools.

Application Profiling

The Win32 SDK provides tools for profiling Win32-based applications on Windows NT. One such tool allows calls to the Win32 API to be profiled to help determine how applications are calling the system and what calls are taking the longest.

The API profiling tool modifies the application's .EXE and .DLLs (if any) so that it dynamically links to a set of profiling DLLs. This is separate from the compile and link process. The API profiling tool, APF32CVT.EXE, is provided with the Win32 SDK and should be used on Windows NT to modify the Win32 binary.

In the \MSTOOLS\WIN32S\PROFILE directory you will find several profiling DLLs: ZERNEL32.DLL, ZSER32.DLL and ZDI32.DLL. These profiling DLLs should be used on Windows 3.1*x* for Win32s profiling. The profiling DLLs can be placed in the same directory as the Win32-based application being profiled.

Once you have verified that profiling is working on Windows NT, you can use the same modified Win32-based application for API profiling on Windows 3.1*x* using the Win32s versions of the profiling DLLs.

Kernel Debugger

The Windows 3.1*x* kernel debugger is fully documented in the Windows 3.1*x* SDK. An updated copy of the debugger (WDEB386.EXE) is provided on the Win32 SDK CD-ROM. The following kernel debugger notes should be sufficient regardless of whether or not you have the Windows 3.1*x* SDK.

Windows NT executables (PE format) support COFF and CV4 debug and symbol information. The Windows 3.1*x* kernel debugger supports a third, older format, SYM. The utility MAPSYMPE.EXE provided in the Win32 SDK creates a separate SYM file from COFF symbol information (which is embedded in the PE file). You can run MAPSYMPE.EXE on Windows NT as the final stage of your build process.

Since WinDbg uses CV4 debug information and MAPSYMPE.EXE uses COFF debug information, it is important to link your application correctly to get the proper debug data for debugging. When creating a debug version of your Win32-based application that will be used with the kernel debugger, specify:

```
-debug:full -debugtype:coff.
```

Specifying **-debugtype:both** creates both CV4 and COFF symbols, but no COFF line numbers are generated; therefore this option is not useful for kernel debugging.

Debugging notes:

1. Link your Win32-based application with **-debugtype:coff** for kernel debugging (WinDbg requires **-debugtype:cv**).
2. Use MAPSYMPE.EXE to generate a .SYM file.

   ```
   mapsympe -o win32app.sym win32app.exe
   ```

   Add a **-n** if **mapsympe** complains about too many line numbers.
3. Invoke the Windows kernel debugger:

   ```
   wdeb386 /c:n /x /s:win32app.sym [/s:win32s.sym ...] \windows\win.com
   ```

   An alternative is to use an input file, as follows:

   Create the file WIN32S.DBG with these contents:

   ```
   /c:1
   /x
   /s:win32app.sym
   /s:win32s.sym
   ...
   \windows\win.com
   ```

   Invoke the debugger:

   ```
   wdeb386 /f:win32s.dbg
   ```

4. The debug version of Win32s provides basic facilities to monitor application execution. These features are triggered by setting bits in a debug flag, and are intended to help in Win32s system development. Some may be helpful for application development, as well.

   Add a line to <WINDOWS-DIRECTORY>\SYSTEM.INI in the [386Enh] section:

   ```
   Win32sDebug=xxxxxxxx
   ```

   where *xxxxxxxx* is a hex value combined with the following flags:

   - 00000001—Verbose. Print messages including notification on application loading and termination, and exceptions.
   - 00000002—Stop on fatal exceptions. Causes the debugger to stop on abnormal exceptions such

as divide by zero, general protection fault, invalid instruction, etc. These exceptions are not necessarily fatal. A general protection fault may occur while Windows checks whether a given hWnd is valid. Even if the exception is abnormal, such as divide by zero, it may be handled "gracefully" by Win32 SEH mechanism (try/except structure). The "Go" command will resume execution.

- 00000004–Stop at 16-bit and 32-bit initialization code, and just before DLL initialization. Allows setting break points in application and DLL initialization routines.
- 00000008–Print message for Win16 APIs that are called. This includes most USER and GDI calls and many KERNEL APIs.
- 00000010–Stop on SetError
- 00000020–Verbose loader
- 00000040–Message return codes displayed before and after thunking
- 00000080–NE resource table information
- 00000100–SEH information
- 00000200–Paging information
- 00000400–Unimplemented/unrecognized message warnings
- 00001000–Trap NULL pointer usage in 32-bit code (similar to 0x2) but for page faults only.
- 00002000–Stop after loading completed. See 0x4.
- 00004000–Trace Virtual Memory Manager in VxD
- 00010000–Pass all int 1 to ring 3 (for debugging hi-level debugger while wdeb386 present)
- 80000000–Pre-load all modules (disable demand paging of modules). With demand paging you may not be able to set a breakpoint if the page is currently not present. It is possible to use 386 debug registers ("br e address") for up to four breakpoints. This debug flag causes all modules to be pre-loaded so code can be disassembled and breakpoints set.

  To assist setting the debug flags, the WIN32SDB.EXE utility (located in the \MSTOOLS\WIN32S\BIN directory) can be used to set/check the state of the debug flags:

{ewc msdncd, EWGraphic, group10904 4 /a "SDKS-5.bmp"}

WIN32SDB.EXE Utility

Setting the following flags makes the following change in SYSTEM.INI:

```
[386Enh]
Win32sDebug=00000022
```

5. All Win32 functions are defined in Win32s, so every Win32-based application can be loaded. Yet, many functions are not supported on Win32s, like security services. Calling these functions will return error values, which depend on the function; the last error is set to ERROR_CALL_NOT_IMPLEMENTED. In debug versions of Win32s, a message with the name of the unsupported function is printed on the debug terminal.
6. Win32s debugging requires a large amount of contiguous virtual memory. You should have at least 4MB of physical memory. You can increase the amount of virtual memory by using the 386 Enhanced applet in the Windows Control Panel. From 386 Enhanced, click the Virtual Memory button and then click the Change button.
7. You can switch between the debug version of Win32s and the end-user (nondebug) version by using the SWITCH.BAT file, which is installed onto the Windows 3.1*x* system using INSTALL.BAT. Run SWITCH for usage information. This script lets you test using the additional debug support provided by the debug version and verifying the application on the end-user version (nondebug). Note: Be sure to run SWITCH.BAT from the directory where this file resides (\WIN32S\BIN).

Shipping Win32s with Win32 Applications

Win32s is not built into Windows 3.1, so software vendors must ship and install Win32s along with the Win32-based application in order to work on Windows 3.1*x*. The key feature of Win32s is that it allows software developers to ship Win32-based applications today for Windows 3.1*x* and Windows NT that will continue to install and work well on future Windows operating systems. Setup programs created today must properly handle various setup issues when installing on current and future operating systems.

The Win32s Setup program provided with the Win32 SDK addresses all the issues raised in this guide. A 16-bit Setup program is required to install Win32s onto Windows 3.1*x*. The sources for the Win32s Setup program are provided. This will allow you to create a custom graphical setup program which addresses the system setup issues correctly, and to save time by re-using an existing solution. The Win32s Setup program is built using the Setup Toolkit shipped with the Microsoft Windows 3.1*x* Software Development Kit, also available separately from Microsoft.

Win32s Installation Rules

For a single Win32-based application that installs correctly on Windows 3.1*x* and Windows NT, a setup program should do the following:

- Install Win32s files on Windows 3.1*x* systems running in Enhanced Mode with paging enabled (paging is enabled by default in a Windows 3.1*x* installation).
- Not install Win32s files if Win32s is already installed (do a version check).
- Install only Win32-based applications on Windows NT and Windows 95 (do not install Win32s files).

These minimum installation requirements ensure that Win32s files are only installed on systems that require them (Windows 3.1*x*) and not on Windows 95 or Windows NT since Windows NT and Windows 95 are capable of running Win32-based applications as they are.

A Win32s setup program should also meet the following criteria:

- Detect Windows 3.0 or lower, provide an informative error message, and not install Win32s.
- Detect Windows 3.1*x* running in standard mode, give an informative error message, and not install Win32s.
- Be prepared for possible future releases of Windows. Win32s should not be installed on any Windows version later than 3.11, since such versions will be able to run Win32-based applications directly.
- Detect the presence of Windows NT and Windows 95 and install only the Win32-based application, not Win32s, since Windows NT and Windows 95 runs Win32 binaries directly.
- If setup detects that Windows NT is running, test whether this is a RISC-based system. If RISC-based, setup should fail with appropriate message (or install R4000 or DEC Alpha version of the Win32-based application).
- Setup should check for the presence of Win32s and only install a later version of Win32s (call **GetWin32sInfo**, exported by W32SYS.DLL and documented in Appendix C). Any future updates to Win32s will have higher version numbers, so that new versions can overwrite earlier releases.
- For OLE applications, install the 16-bit OLE libraries in compliance with the OLE Programmer's Reference. This implies that each 16-bit DLL is overwritten only by newer 16-bit DLLs. Install the 32-bit OLE libraries in the same manner.

While the list of setup issues above seems long, a simple set of tests can detect these conditions. It is necessary to use a 16-bit installation program to install Win32s since Win32s isn't necessarily running yet. It is recommended that you use a Win16 program (rather than MS-DOS) and preferably based on the Win32s Setup program that ships with the Win32 SDK. Several functions can be called from a Win16 program to determine the version of the system, version of Win32s and the platform:

- **GetVersion**: Determine version number of Windows (for MS-DOS), Windows NT, and past and future versions of Windows.
- **GetWinFlags**: Flag indicates whether the Win16-based application is running on Windows NT. Test for 0x00004000: if set, then running on Windows NT.
- **_environ**: Use this C run-time function to determine if on a RISC-based system. Windows NT runs Win16-based applications using an emulator on RISC systems. Therefore, Setup should first determine if the system is Windows NT using **GetWinFlags** (as discussed above), then check to ensure that the environment variable PROCESSOR_ARCHITECTURE is equal to INTEL before installing an *x*86-based Win32 application (or install the appropriate RISC binary).

Win32s 1.1 added an exported function from W32SYS.DLL to make it easy to determine the version number of Win32s that is installed. W32SYS.DLL is a 16-bit DLL, and this function should be called from a 16-bit Windows Setup program.

Before shipping a Win32-based application, it is critical to test your application fully on Windows 3.1*x*,

Windows for Workgroups, Windows 95, and Windows NT.

Win32s File Installation and Configuration

This section lists all Win32s files and the directories in which the files must reside. The complete set of files must be installed. Partial installation will affect other Win32-based applications that rely on Win32s to run. All Win32s files contain version information. Your setup program must version-check every file and only install the most recent version on a file-by-file basis.

Note Win32s setup programs must implement version checking. There are previously released versions of Win32s. Only the most recent version of Win32s should be installed. Earlier versions of Win32s, such as Win32s 1.0, must not be installed over a Win32s 1.25 installation.

The following set of Win32s files must be installed in the <SYSTEM> directory (where <SYSTEM> is the drive and directory returned from **GetSystemDirectory**. Note that this may not be on the same drive as the Windows directory.)

```
OLECLI.DLL
W32SYS.DLL
WIN32S.INI*
WIN32S16.DLL
WINMM16.DLL
```

The WIN32S.INI file is not shipped as part of Win32s but should be created during setup as discussed in the next section.

Note The WINMM16.DLL file and the driver for After Dark™ (ADWRAP.DRV) do not coexist. After installing Win32s, look at WIN32S.INI and make certain that the ADWRAP.DRV is listed in the device= line *after* WINMM16.DLL and MMSYSTEM.DLL. Otherwise problems may result.

The following set of Win32s files must be installed in the <SYSTEM>\WIN32S directory:

.EXE file:

```
 WIN32S.EXE
```

.DLL files:

```
 COMDLG32.DLL
COMCTL32.DLL
CRTDLL.DLL
LZ32.DLL
NETAPI32.DLL
NTMSG.DLL
OLECLI32.DLL
OLESVR32.DLL
SCK16THK.DLL
SHELL32.DLL
VERSION.DLL
W32SCOMB.DLL
W32SKRNL.DLL
WINMM.DLL
WSOCK32.DLL
```

Configuration files and drivers:

```
W32S.386
WINSPOOL.DRV
```

.NLS files:

You must also install National Language Support (.NLS) files that correspond to the market of the Win32s application. For a Win32s application that has an international market, you must install all of the .NLS files found in \MSTOOLS\WIN32S\NLS. For example, a Win32s application that is released only in the United States must install P_437.NLS, P_850.NLS, and P_1252.NLS.

The following files must be removed from the <SYSTEM>\WIN32S directory if an earlier version of Win32s is present:

```
ADVAPI32.DLL
C_*.NLS
EM87.DLL
GDI32.DLL
KERNEL32.DLL
MPR.DLL
NTDLL.DLL
USER32.DLL
```

See Installing OLE Applications for OLE installations.

Once the files are installed, enable Win32s by loading the Win32s VxD. As with any VxD, do this by adding a *device*= reference in the [Enh386] section of the Windows SYSTEM.INI file:

```
[Enh386]
device=c:\windows\system\win32s\w32s.386
```

Enable multimedia wave callbacks by having the WINMM16.DLL loaded when Windows boots. Do this by using the [Boot] section of SYSTEM.INI file. Add WINMM16.DLL to the *device*= line as below:

```
[Boot]
device=mmsytem.dll winmm16.dll
```

Please refer to the License Agreement concerning Win32s Redistributable Files.

Installing OLE Applications

This guide discusses issues related to the distribution of your OLE application. It continues with a description of the files that must be included on your distribution disk(s), discusses special considerations for OLE application distribution, and concludes with a discussion of special considerations for distribution of OLE server applications.

Disk Contents

You must distribute the files listed following if your application is OLE-enabled. When installing your application on the end user's system your install/setup program *must* correctly install all of the following files to the user's <SYSTEM> directory:

COMPOBJ.DLL
OLE2.DLL
OLE2PROX.DLL
OLE2DISP.DLL
OLE2NLS.DLL
OLE2CONV.DLL
OLE2.REG
STDOLE.TLB
STORAGE.DLL
TYPELIB.DLL

You must also correctly install all of the following files to the user's <SYSTEM>\WIN32S directory:

OLE32.DLL
OLEAUT32.DLL
OLE2THK.DLL
STDOLE32.TLB

Note The installation of 32-bit OLE components must occur if the 16-bit OLE components are not installed. The 16-bit OLE files may already be present on the user's system.

Since OLE requires all of these files in order to run correctly, your setup/installation program must install *all of them* regardless of whether or not the application itself uses the files.

Where to Install

Use the **GetSystemDirectory** function to get the name of the user's <SYSTEM> directory. The install/setup program *must* check the user's <SYSTEM> subdirectory to verify if copies of the OLE files are already present. If the files are already present in the user's system, the version information of *each* file *must* be checked to make sure that newer versions of the OLE files will not be overwritten by older versions.

You can use the File Installation Library, VER.DLL, supplied with Microsoft Windows Version 3.1*x*, to extract a version string from the application. For more information on VER.DLL, see the *Microsoft Windows Programmer's Reference, Volume 1: Overview* in Version 3.1*x* of the Microsoft Windows Software Development Kit.

Additionally, the application should copy its registration entry file *YYYY*.REG (*YYYY* is determined by the object application) into the directory where the application installs its executable (.EXE file). This will allow the user to double-click on the *YYYY*.REG file and re-register the application in case the registration database (REG.DAT) gets corrupted. For the content and creation of the registration files, see Registering OLE Libraries.

Checking the TEMP Entry

The setup of an application should detect if the user's TEMP= entry points to a removable media and prompt the user to change it to point to a local hard drive. (If the application is doing AUTOEXEC.BAT modification, it should make the modification for the TEMP= entry also.)

Checking the SHARE.EXE

In order to function, the OLE Compound File implementation requires that file range-locking be enabled. If range-locking is not already installed on the machine, then you must add the MS-DOS command

SHARE.EXE

to AUTOEXEC.BAT. SHARE.EXE is the common way that range-locking is supported on Microsoft Windows 3.1. On Microsoft Windows for Workgroups, range-locking is part of the system (through VSHARE.386) and SHARE.EXE should not be installed.

Caution The SHARE.EXE entry should follow the PATH statement in AUTOEXEC.BAT, but precede any statement that starts Windows or otherwise prevents SHARE.EXE from being executed.

To determine if range-locking is installed, open a local file and attempt to acquire a range lock on it. If an error is returned, then range-locking is not supported (see MS-DOS Int 21h Function 5Ch for further details). AUTOEXEC.BAT is a suggested local file to use for this purpose.

If range-locking is not supported, add SHARE.EXE to AUTOEXEC.BAT. If range-locking is in fact currently supported, you may still need to modify AUTOEXEC.BAT. Scan AUTOEXEC.BAT for SHARE.EXE. If it is not found, do nothing; if it is found, then ensure that its /L and /F options are at least as large as outlined below.

The **/L** option on the MS-DOS SHARE command indicates the number of locks that can be placed on files. The **/F** parameter allocates file space (in bytes) for the MS-DOS storage area used to record file-sharing information. The default setting for the **/L** option is 20 locks; the default setting for the **/F** option is 2048. Each open OLE Compound File requires for its own use four (or fewer) dedicated locks. The Compound File implementation requires a small number (fewer than five) of additional locks, independent of how many files are open. Each lock reserved by **/L** requires 18 bytes.

It is recommended that an **/L** setting of at least **/L:500** and an **/F** setting of at least **/F:5100** be used:

SHARE /L:500 /F:5100

If the **/L** or the **/F** setting to SHARE has a larger value than listed here, installation programs should leave the setting as it is rather than reducing it.

Note Be sure to add the necessary copyright notice to the copyright notice(s) for your application. See the SDK documentation for information on copyright notices.

Registering OLE Libraries

The OLE libraries require that many internal interfaces be registered in the registration database. Therefore, if your installation program installs any of the OLE libraries on a machine that does not have them already, it should register OLE specific information by running

```
REGEDIT.EXE /S OLE2.REG
```

to register the OLE internal interfaces in the REG.DAT file.

Also, if your installation/setup program has overwritten older versions of the application on the user's hard drive, run the **REGEDIT.EXE /S OLE2.REG** entry mentioned above.

If your installation program does not install OLE libraries because it found more recent libraries on the user's drive, it should not register the OLE interface information.

Note When checking the version stamp (using VER.DLL) on existing OLE libraries to determine whether to replace them on the user's hard drive, do so on a file-by-file basis.

If OLE2.REG is registered, copy the file to the user's <SYSTEM> directory.

Win32s Setup Sample

As part of the Win32s development files, a sample setup project is provided that will recreate the Win32s Setup floppy-disk image, which is also provided on the Win32 SDK CD-ROM. After installing Win32s, you will find a \WIN32S\SETUP directory containing Microsoft Setup Toolkit binaries, floppy disk layout files, and sources to Setup extensions that are used to install Win32s. You are encouraged to use these Setup files to build your own Win32-based application and Win32s installation program. Use the *Setup Toolkit for Windows* manual as a reference.

You will find the following Win32s Setup Sample files:

- \WIN32S\SETUP contains the layout files and required Setup Toolkit binaries.
- \WIN32S\SETUP\BLDCUI contains the sources and dialog templates for Win32s Setup screens.
- \WIN32S\SETUP\INIUPD contains the Setup extension DLL that:
  1. Determines if Win32s is already installed.
  2. Calls **GetWin32sInfo** exported by W32SYS.DLL to determine the version of Win32s already on the user's computer, if any.
  3. Adds the VxD entry to SYSTEM.INI.
- \WIN32S\SETUP\BUILD.BAT is used build the floppy-disk image in \WIN32S\DISKS\RETAIL\ FLOPPY.

The FreeCell sample consists of three files: FREECELL.EXE, FREECELL.HLP, and CARDS.DLL. To ship your own application, modify the following setup sample files and change the FreeCell references as appropriate for your application:

- For the Win32s core, W32S.LYT contains the names of the files to install, destination directories, and whether the files are compressed or not. This is the input file for the DSKLAYT2 utility, which creates the setup disk layout. For Win32s plus OLE, use OW32S.LYT.
- 32INST.MST is an MS-Test script that controls the setup process events. This file contains strings to add a Program Group to the Window Program Manager.
- DIALOGS.DLG in the BLDCUI directory contains the dialog definitions and text for the Setup screens.

The Win32s Setup program creates/updates a WIN32S.INI file in the \WINDOWS\SYSTEM directory with the following information:

```
[Win32s]
Setup=1
Version=1.2.0.0
[Nls]
AnsiCP=1252
[FreeCell]
Setup=1
Version=1.2.0.0
```

The [Win32s] section indicates whether Win32s is fully installed (Setup=1 when Win32s is correctly installed.) This section should list the current version of Win32s that is installed. The [Nls] section tells Win32s which code page should be used for translation. If the [Nls] section is not present, code page 1252 is used by default. See **International Support** for more information.

When OLE is installed a corresponding [ole] section appears in WIN32S.INI.

The WIN32S.INI file can also be used by Win32-based applications to indicate what Win32-based applications have been installed, and what version of Win32s was installed at the time the application was installed. FreeCell is listed above as an example of a Win32-based application entry. The example indicates that Win32s version 1.25 was installed.

The Win32s Setup program requires the 16-bit Microsoft Setup Toolkit, which is a separate product available from Microsoft that was previously bundled with the Microsoft® Windows 3.1 SDK.

Note The disk layout utility that ships with the Microsoft Setup Toolkit uses COMPRESS.EXE for file compression. Several versions of this utility are shipped with other products. Be sure that the version shipped with the Microsoft Setup Toolkit is first in the PATH statement when building the Win32s Setup sources.

Unsupported Features

The following features are planned for Win32s version 1.3:

- Windows 95/Windows NT 3.51 Common Controls
- Windows 95/Windows NT 3.51 Common Dialogs
- WinHelp 4.0
- OLEDLG (OLE Dialogs)

Win32s 1.3 is planned to be the last release of Win32s as Windows 95 replaces Windows 3.1 as the volume platform.

The following Win32 features are not currently planned for Win32s:

- MAPI
- RPC
- Console APIs
- Unicode APIs (Win32s does support Code Page/Unicode translation APIs)
- Security APIs
- Comm APIs
- Asynchronous File I/O
- Threads
- Paths (graphics object)
- Enhanced Metafiles
- Bezier curves

While it is impossible to include all of the features listed above in a future Win32s release, your feedback on desired features is very important. Please post your suggestions on the CompuServe® MSWIN32 forum in the *API-Win32s* section.

System Limits

This guide lists Windows 3.1*x* and Windows NT differences that programmers should be aware of when designing Win32-based applications to run on both platforms. Reviewing these differences can save time developing and debugging applications.

Window Manager (User)

Win32s relies on Windows 3.1*x* to provide standard dialog controls such as list boxes, combo boxes, and edit controls. Win32s translates messages between the controls and the Win32-based application.

Controls have size limitations that do not exist on Windows NT. An edit control, for example, is limited to somewhat less than 64K of text; list boxes store data in one 64K heap. Therefore, a Win32-based application can create a large file by way of an edit control in Windows NT and not be able to read the file back into the same edit control when the Win32-based application is run with Windows 3.1*x* or Windows 95.

Controls support specifying limits on the amount of information they will hold, such as EM_SETLIMITTEXT for edit controls. One solution is to specify a lowest common denominator for controls to ensure compatibility across platforms. Other solutions are application specific.

Win32s does not support EM_SETHANDLE and EM_GETHANDLE. These messages allow the sharing of local memory handles between an application and the system. In Win32s an application's local heap is 32 bits and allocated from the global heap, so Windows 3.1*x* cannot interpret a local handle. Text for multiline edit controls is stored in the 16-bit local heap. This means that the amount of edit control text is limited to somewhat less than 64K. To read and write multiline edit control text, an application should use the WM_GETTEXT and WM_SETTEXT messages.

Any private application message must be defined above WM_USER + 0x100. This will ensure that there is no collision between private messages and dialog control messages on Windows 3.1*x*.

When calling the **PeekMessage** function, a Win32-based application should not filter any messages for Windows 3.1*x* private window classes (button, edit, scroll bar, and so on). The messages for these controls are mapped to different values in Win32, and checking for the necessity of mapping is a time-consuming operation.

Win32s child window identifiers must be sign-extended 16-bit values. This precludes using 32-bit pointer values as child window identifiers.

Windows NT dynamically grows the message queue size. Windows 3.1*x* has a default message queue size of 8, which can be changed by calling **SetMessageQueue**. This call is supported in Win32s and is a no-operation instruction on Windows NT.

The **SetClipboardData** function must be called with global handles to ensure that the data can be accessed by other applications.

Applications specify the return value for the **DialogBox** function by way of the *nResult* parameter to the **EndDialog** function. This parameter is of type **int**, which is 32-bits for Win32-based applications. However, this value is thunked by Win32s to the Windows 3.1*x* **EndDialog** function, which will truncate it to 16 bits. Win32s sign-extends the return code from **DialogBox**.

The **TranslateMessage** function is better supported on Windows NT than on Windows 3.1*x*. Win32s provides the Windows 3.1*x* functionality that is to return TRUE when any key or number is typed (which generates a KEYDOWN/KEYUP sequence). On Windows NT, function and arrow keys will also result in **TranslateMessage** returning TRUE. Few applications test the return value from **TranslateMessage**.

Resource IDs (identifiers in resource files) must be 16-bit values.

The resource table size is limited to 32K (not including the resource data).

Graphics (GDI)

Windows 3.1*x* has a limit of five cached device contexts (DCs). The **GetDC** call obtains a cached DC in Windows 3.1*x*; therefore, it is important to call **ReleaseDC** before checking the message queue. Otherwise, another application may be scheduled and there will be one less cached DC for all applications in the system. Windows NT does not have a cached DC limit but applications should not waste DCs. It is important to follow the **GetDC/ReleaseDC** model for compatibility on both platforms and good memory management.

Windows 3.1*x* allows drawing objects (pens and brushes) to be deleted while still selected in a DC. However, the memory allocated for the drawing object remains allocated until the process terminates. In Windows NT, the **DeleteObject** call fails. Don't delete selected objects, for this may slowly use up system resources while the application is running on Windows 3.1*x*.

The **SetDIBitsToDevice** function is not supported for bitmaps in memory DCs, and is limited to bitmaps of up to 3.75MB when using block-transfers (BLTs) to update the display. This limit was chosen to support a 1280x1024x24 bitmap. Since the limit is one of size and not resolution, larger bitmaps with a lower color depth are supported (for example, 1600x1200x16).

Process cleanup assures that all objects allocated by an application exiting the system are destroyed and the memory freed for reuse by the system or other applications. Windows 3.1*x* added additional robustness and process cleanup compared to Windows 3.0, but is not as complete as Windows NT. Win32-based applications running on Windows 3.1*x* must abide by the same rules as 16-bit Windows applications and properly free objects allocated while running, especially pens, brushes and other graphics objects.

Windows 3.1*x* has a 16-bit world coordinate system (as does Windows 95). Windows NT supports a 32-bit world coordinate system. You must use the Windows 3.1*x* limit in developing a Win32-based graphics application that will run on Windows 3.1*x*, Windows 95, and Windows NT.

Windows NT allows Win32-based applications to call **SetCursor** and specify an icon handle. Windows NT supports color icons and color cursors; Windows 3.1*x* only supports color icons and monochrome cursors. Therefore, you cannot pass an icon handle to **SetCursor** on Win32s.

In Win32s, **GetIconInfo** sets to TRUE the **fIcon** member of the **ICONINFO** structure pointed to by *pIconInfo*. In Windows NT and Windows 95, **GetIconInfo** sets **fIcon** to FALSE for a cursor and TRUE for an icon.

For an application's icon in Program Manager, Windows NT uses the first icon specified in the RC file; Windows 3.1 uses the lowest-numbered icon. To use the same icon in Windows NT and Windows 3.1, the first icon in the RC file should be the lowest-numbered icon.

System Services (Kernel)

Win32s processes that are started by Win16-based applications should be launched using **WinExec**. The Win16 version of **LoadModule** will not start a Win32s process. If you are using the Win32 version of **LoadModule**, note the following difference in specifying the **lpCmdLine** in the **LOADPARAMS32** structure passed to this function:

- On Windows NT, **lpCmdLine** points to a Pascal-style string that contains a correctly-formed command line. The first byte of the string contains the number of bytes in the string. The remainder of the string contains the command line arguments, excluding the name of the child process. If there are no command line arguments, this parameter must point to a zero length string; it cannot be NULL.
- On Win32s, **lpCmdLine** points to a null-terminated string that contains a correctly-formed command line. The string must not exceed 120 bytes in length.

The **VirtualAlloc** function provides a feature on Windows NT allowing the process to request the memory allocation at a specified virtual address. This is supported on Win32s, but applications should not be designed to depend on available address range. Windows NT allocates memory in the low 2GB address space, Win32s allocates memory in the high 2GB address space. Applications can query the address space using **GetSystemInfo**.

Do not specify GMEM_FIXED when using **GlobalAlloc** (use GMEM_MOVEABLE instead). On Win32s, this will result in locked pages in memory. Therefore, allocations will be limited to physically available memory rather than virtual limits (pageable memory allocations).

The implementation of the Win32 **Sleep** function differs in Win32s and Windows NT. On Win32s, this function calls **Yield**, which will return immediately if a) no other applications have messages in their message queues awaiting processing or b) the application calling **Yield** still has messages in its queue. It is not possible to block for the specified delay time (as occurs on Windows NT) since Windows 3.1*x* is a cooperative multitasking system, not a pre-emptive multitasking system. Time delays should be implemented in **PeekMessage** loops with calls to **GetSystemTime** to control delay time.

On Windows NT you cannot call **DeleteFile** and delete an open file (for normal I/O or as a memory mapped file). On MS-DOS (and therefore Win32s) you can, even if SHARE.EXE is loaded. Deleting such a file may result in loss of data and application failure. Be very careful when using **DeleteFile** to ensure that the file is not in use.

When a file is created or modified, the precision of the recorded time is ±2 seconds (this is an MS-DOS limitation from the FAT file system). Also, file times returned in Win32s are always in local time, not UTC.

The **GetPrivateProfileSection** and **GetProfileSection** functions work only on initialization files that have unique keys. For example, in the following private initialization file:

```
[TestSection]
Entry1=123
Entry1=456
Entry1=789
Entry2=ABC
```

The **GetPrivateProfileSection** function will return three copies of the string: Entry1=123.

Windows 3.1*x* supports a single value (string) per key in its registration database. Windows NT supports a multivalue per key registration database and new APIs to manipulate the database. Win32s supports the subset of Win32 registry APIs that map to Windows 3.1*x* functionality.

Win32-based applications and DLLs should be linked with 4K alignment. This is the default for the LINK32 linker that ships with the Win32 SDK. 64K alignment is supported, but 4K is more efficient for the memory manager in the shared 2GB address space of Windows 3.1*x*.

Calling **LoadLibrary** on a 16-bit DLL will succeed and return a valid *hModule*, but **GetLastError** will return ERROR_BAD_EXE_FORMAT. This was done to support the **LoadLibrary**/**GetProcAddress** method of printing, and for some resource APIs. The *hModule* returned cannot be used for other methods. Make sure that **FreeLibrary** is called for this *hModule* when you are finished with the handle.

Networking

The WinSocket API supports blocking and non-blocking calls. The WinSocket specification indicates that blocking calls should be avoided on systems (such as Windows 3.1*x*) that are non-pre-emptive. A portable Win32 WinSocket application should use non-blocking calls in order to run well on Win32s, Windows 95, and Windows NT.

Windows NT supports certain **NetBios** features that are not supported on Win32s due to the non-pre-emptive shared memory design of Windows 3.1*x*. The **ncb_event** member of the **NCB** structure is ignored since Win32s does not implement Win32 events. Also, Windows NT maintains a per-process name table; Win32s maintains one system-wide name table. Before calling **Netbios** with NCBRESET, there should be no outstanding asynchronous messages – all such messages should be received or canceled.

Multimedia

Win32s supports all multimedia sound APIs provided on Windows NT except for MIDI callbacks, so the CALL_FUNCTION flag of **midiOutOpen** and **midiInOpen** is not supported. Also, Multimedia event callbacks (**timeSetEvent** and **timeKillEvent**) are not supported.

Win32s does not support the SND_ALIAS, SND_FILENAME, and SND_NOWAIT flags for **PlaySound**. These flags are supported in Windows NT and Windows 95.

Win32s API Reference

The functions listed in this guide are supported by Win32s and define interfaces for Setup programs and the Universal Thunk.

GetWin32sInfo

WORD GetWin32sInfo(
LPWIN32SINFO *lpInfo*
);

Parameters

lpInfo
Points to a WIN32SINFO structure, defined as follows:

```
typedef struct {
    BYTE bMajor;
    BYTE bMinor;
    WORD wBuildNumber;
    BOOL fDebug;
} WIN32SINFO, far * LPWIN32SINFO;
```

Return Value

The return value is zero if the function is successful. Otherwise, the possible values are:

1 Win32s VxD not present.
2 Win32s VxD not loaded because paging system not enabled.
3 Win32s VxD not loaded because Windows is running in standard mode.

Remarks

Function is exported by W32SYS.DLL in Win32s version 1.1 and later. The function fills the specified structure with the information from Win32s VxD. A 16-bit Windows Setup program should use this function to determine if Win32s is already installed version and only install a later Win32s release. The Setup program must use **LoadLibrary** and **GetProcAddress** to call the function since the function did not exist in Win32s 1.0.

Example

```
// Indicates whether Win32s is installed and version number
// if Win32s is loaded and VxD is functional.

BOOL FAR PASCAL IsWin32sLoaded(LPSTR szVersion)
{
    BOOL              fWin32sLoaded = FALSE;
    FARPROC           pfnInfo;
    HANDLE            hWin32sys;
    WIN32SINFO        Info;

    hWin32sys = LoadLibrary("W32SYS.DLL");

    if (hWin32sys > HINSTANCE_ERROR) {
        pfnInfo = GetProcAddress(hWin32sys, "GETWIN32SINFO");
        if (pfnInfo) {
            // Win32s version 1.1 (or later) is installed
            if (!(*pfnInfo)((LPWIN32SINFO) &Info)) {
                fWin32sLoaded = TRUE;
                wsprintf(szVersion, "%d.%d.%d.0",
                         Info.bMajor, Info.bMinor, Info.wBuildNumber);
            } else
```

```
                fWin32sLoaded = FALSE;    // Win32s VxD not loaded.
        } else {
            // Win32s version 1.0 is installed.
            fWin32sLoaded = TRUE;
            lstrcpy( szVersion, "1.0.0.0" );
        }
        FreeLibrary( hWin32sys );
    } else                                // Win32s not installed.
        fWin32sLoaded = FALSE;

    return fWin32sLoaded;
}
```

Universal Thunk API

The universal thunk (UT) allows a Win32-based application to call a routine in a 16-bit DLL. It also allows a 16-bit routine to call back to a 32-bit function. This API represents the means by which these calls and associated operations can be completed.

UTRegister

BOOL UTRegister(
  HANDLE *hModule*,
  LPCTSTR *lpsz16BITDLL*,
  LPCTSTR *lpszInitName*,
  LPCTSTR *lpszProcName*,
  UT32PROC **ppfn32Thunk*,
  FARPROC *pfnUT32CallBack*,
  LPVOID *lpBuff*,
);

This routine registers a universal thunk (UT) that can be used to access 16-bit code from a Win32 application running via Win32s on Windows 3.1. Only one UT can be created per Win32 DLL. The thunk can be destroyed by calling **UTUnRegister**. **UTRegister** will automatically load the 16-bit DLL specified. After loading the 16-bit DLL, Win32s calls the initialization routine. Win32s creates a 32-bit thunk that is used to call the 16-bit procedure in the 16-bit DLL.

Parameters

hModule
Handle of 32-bit DLL. The UT provides a mechanism to "extend" a Win32 DLL into Windows 3.1. The thunk is owned by the DLL and every DLL is limited to one thunk.

lpsz16BITDLL
Points to a null-terminated string that names the library file to be loaded. If the string does not contain a path, Win32s searches for the library using the same search mechanism as **LoadLibrary** on Windows 3.1.

lpszInitName
Points to a null-terminated string containing the function name, or specifies the ordinal value of the initialization function. If it is an ordinal value, the value must be in the low word and the high word must be zero. This parameter can be NULL if no initialization or callback is required.

lpszProcName
Points to a null-terminated string containing the function name, or specifies the ordinal value of the 16-bit function. If it is an ordinal value, the value must be in the low word and the high word must be zero.

ppfn32Thunk
Return value is a 32-bit function pointer (thunk to 16-bit routine) if **UTRegister** is successful. This function can be used to call the 16-bit routine indirectly.

pfnUT32CallBack
Address of the 32-bit callback routine. Win32s creates a 16-bit callable thunk to the 32-bit function and provides it to the initialization routine. The 32-bit routine does not need to be specified as an EXPORT function in the DEF file. No callback thunk is created if either this parameter or *lpszInitName* is NULL.

lpBuff
Pointer to globally allocated shared memory. Pointer is translated into 16:16 alias by UT and is passed to the initialization routine. This parameter is optional and ignored if NULL.

Return Value

Function returns TRUE if the DLL is loaded and **UT16Init** routine was successfully called or FALSE if an error occurred. To obtain extended error information, call **GetLastError**. Typical errors are:

ERROR_NOT_ENOUGH_MEMORY
ERROR_FILE_NOT_FOUND
ERROR_PATH_NOT_FOUND

ERROR_BAD_FORMAT
ERROR_INVALID_FUNCTION
ERROR_SERVICE_EXISTS (when the UT is already registered).

Remarks

Registering the UT enables two capabilities for communicating between 32-bit and 16-bit routines. The first capability is to allow a Win32 application to call a 16-bit routine passing data using globally shared memory. This is a Win32 application initiated data transfer mechanism. The second capability is to register a callback routine by which 16-bit code can callback into a 32-bit routine in a Win32 DLL. Again, shared global memory is used to transfer data.

UTRegister will result in Win32s loading the specified DLL with normal Windows 3.1 DLL initialization occurring. Win32s will then call the initialization routine passing this function a 16:16 thunk to the 32-bit callback function. The initialization routine can return data in a global shared memory buffer. The initialization routine must return a nonzero value to indicate successful initialization.

The initialization routine must return a nonzero value to indicate successful initialization. If it fails no thunk is created.

UTRegister returns a 32-bit thunk to the 16-bit **UT16Proc**. This pointer can be used in Win32 code to call the 16-bit routine:

```
dwUserDefinedReturn = (*pfnUT16Proc)(pSharedMemory, dwUserDefinedSend,
        lpTranslationList);
```

The UT will translate the 32-bit linear address of the shared memory to a 16:16 segmented pointer, along with all the addresses listed by *lpTranslationList* before passing it to the 16-bit **UT16Proc** routine.

The 16-bit code can call Win32 code using the callback mechanism:

```
dwUserDefinedReturn = (*pfnUT32CallBack)(pSharedMemory,
        dwUserDefinedSend, lpTranslationList);
```

The 32-bit callback routine should be defined as follows (but does not need to be exported in the DEF file):

```
DWORD WINAPI UT32CBPROC(LPVOID lpBuff, DWORD dwUserDefined);
```

See Also

UT16Proc, **UTUnRegister**, **LoadLibrary** (16-bit version), **GetProcAddress** (16-bit version)

UTUnRegister

VOID UTUnRegister(
  HANDLE *hModule*
);

Requests Win32s to call **FreeLibrary** for the 16-bit DLL loaded by **UTRegister**. Also, destroys the dynamically created universal thunk (UT).

Parameters

hModule
Handle of 32-bit DLL which previously registered the UT.

Return Value

No return value.

Remarks

This call allows the single dynamically created UT to be destroyed and the 16-bit DLL dereferenced. Win32s will clean up these resources automatically when the Win32 DLL is freed (normally or abnormally).

See Also

UTRegister, **FreeLibrary** (16-bit version)

UT16INIT

DWORD FAR PASCAL UT16INIT(
  UT16CBPROC *pfnUT16CallBack,*
  LPVOID *lpBuff*
);

UT16INIT name is a placeholder for the application-defined function name. The actual name must be exported by including it in the EXPORTS statement in the DLL's DEF file. This function will be called only once upon calling **UTRegister** by the 32-bit DLL.

Parameters

pfnUT16CallBack
16-bit thunk to 32-bit callback routine, as specified at registration.

lpBuff
Pointer to general purpose memory buffer that was passed by the 32-bit code.

Return Value

1 Upon success.

0 Upon failure. If the initialization function fails, no stub will be created and **UTRegister** will return 0.

See Also

UTRegister

UT16CBPROC

typedef DWORD FAR PASCAL(
  LPVOID *lpBuff*,
  DWORD *dwUserDefined*,
  LPVOID **lpTranslationList*,
);

Prototype of the 16-bit callable stub that is called by the 16-bit DLL. The stub will translate the *lpBuff* pointer to a 32-bit pointer, scan the translation list and translate all its pointers to 32-bit pointers, arrange the stack as a 32-bit stack, and then call the 32-bit procedure in the 32-bit DLL.

Parameters

lpBuff
Pointer to general-purpose memory buffer. This segmented pointer is translated to flat address and passed to the 32-bit callback procedure. This parameter is optional. If not used, should be NULL.

dwUserDefined
Available for application use.

lpTranslationList
A far (16:16) pointer to an array of far (16:16) pointers that should be translated to flat form. The list is terminated by a null pointer. No validity check is performed on the address except for null check. The translation list is used internally and not passed to the 32-bit callback procedure. This parameter is optional.

Return Value

User defined

See Also

UTRegister

UT32CBPROC

DWORD WINAPI UT32CBPROC(
  LPVOID *lpBuff*,
  DWORD *dwUserDefined*
);

UT32CBPROC name is a placeholder for the application-defined function name. It does not have to be exported. That callback function will be called indirectly by the 16-bit side.

Parameters

lpBuff
Pointer to general purpose memory buffer as passed to 16-bit callback thunk.

dwUserDefined
Available for application use.

Return Value

User defined.

Remarks

Memory allocated by 16-bit code and passed to 32-bit must first be fixed in memory.

See Also

UTRegister

UT16PROC

DWORD FAR PASCAL UT16PROC(
  LPVOID *lpBuff*,
  DWORD *dwUserDefined*
);

UT16PROC name is a placeholder for the application-defined function name. The actual name must be exported by including it in the EXPORTS statement in the DLL's DEF file.

Parameters

lpBuff
Pointer to general purpose memory buffer that is passed by the 32-bit code.

dwUserDefined
Available for application use.

Return Value

User defined.

See Also

UTRegister

UT32PROC

typedef DWORD (
  LPVOID *lpBuff***,**
  DWORD *dwUserDefined***,**
  LPVOID **lpTranslationList*
);

Prototype of the 32-bit callable stub that is called by the 32-bit DLL. The stub will translate the *lpBuff* pointer to a 16:16 pointer, scan the translation list and translate all its pointers to 16:16 pointers, arrange the stack as a 16-bit stack, and then call the 16-bit procedure in the 16-bit DLL.

Parameters

lpBuff
Pointer to general purpose memory buffer. This 32-bit pointer is translated to 16:16 form and passed to the 16-bit procedure. The segmented address provides addressibility for objects up to 32K. This parameter is optional.

dwUserDefined
Available for application use.

lpTranslationList
Pointer to an array of pointers that should be translated to segmented form. The list is terminated by a null pointer. No validity check is performed on the address except for null check. The translation list is used internally and not passed to the 16-bit procedure. This parameter is optional and if not used should be NULL.

Return Value

User defined.

See Also

UTRegister

Universal Thunk Auxiliary Services

The following are address translation services available for 16-bit code only.

UTSelectorOffsetToLinear
UTLinearToSelectorOffset

UTLinearToSelectorOffset

LPVOID UTLinearToSelectorOffset(
  DWORD *lpByte*,
);

Translate a flat address to a segmented form.

Parameters

lpByte
  Translate a flat address to segmented form.

Return Value

Equivalent segmented address (16:16 far pointer).

Remarks

The returned address guarantees addressibility for objects up to 32K.

See Also

UTSelectorOffsetToLinear, **GlobalFix**

UTSelectorOffsetToLinear

DWORD UTSelectorOffsetToLinear(
  LPVOID *lpByte*;
);

Parameters

lpByte
Translate a segmented address to flat form.

Return Value

Equivalent flat address.

Remarks

The base of the flat selectors used by Win32s process is not zero. If the memory is allocated by 16-bit API it must be fixed before its address can be converted to flat form.

See Also

UTLinearToSelectorOffset

Legal Notice

Olivetti is a registered trademark of Ing. C. Olivetti.

PostScript is a registered trademark of Adobe Systems, Inc.

R4000 is a trademark of MIPS Computer Systems, Inc.

Roland is a registered trademark of Roland Corporation.

SCSI is a registered trademark of Security Control Systems, Inc.

Epson is a registered trademark of Seiko Epson Corporation, Inc.

Silicon Graphics is a registered trademark and OpenGL is a trademark of Silicon Graphics, Inc.

Stacker is a registered trademark of STAC Electronics.

Tandy is a registered trademark of Tandy Corporation.

Unicode is a registered trademark of Unicode, Incorporated.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company, Ltd.

VAX is a trademark of Digital Equipment Corporation

Yamaha is a registered trademark of Yamaha Corporation of America.

Paintbrush is a trademark of Wordstar Atlanta Technology Center.

Microsoft Win32 Developer’s Reference

You have requested information from the **Microsoft Win32 Developer’s Reference**. One or more of these help files is not available on your system.