🔶 Win32s 1.1 Programmer's Reference

Copyright

CHAPTER 1   **Introduction**

Microsoft® Win32s(TM) version 1.1 is an operating system extension that allows Win32(TM) applications for Microsoft® Windows NT(TM) to run on Microsoft® Windows(TM) version 3.1 and Microsoft® Windows for Workgroups(TM).

Win32s offers software developers:

- A 32-bit programming model for Windows 3.1 which is binary compatible with Windows NT
- 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
- An established market of Windows 3.1 systems and the new Windows NT market to sell Win32 applications
- Ship a single Win32 product for both Windows NT and Windows 3.1

Win32s consists of a virtual device driver (VxD) and a set of dynamic link libraries (DLLs) that extend Windows 3.1 to support Win32 applications. The Win32s files must be shipped with the Win32 application and installed on the Windows 3.1 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 system
- Development files as part of the Win32 SDK to create, debug, test, and ship your Win32 application and Win32s files.

The Win32 SDK relies on Windows NThosted development tools, such as the compiler and debugger, to build and test your Win32 application.

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

This chapter provides Win32s installation instructions using a setup program provided on the Win32 SDK CD-ROM. The remainder of the manual covers issues developers should consider when writing Win32 applications targeted for both Windows NT and Windows 3.1.

Win32s System Requirements

To install Win32s and run Win32 applications, you need:

- A system running Microsoft Windows for Workgroups or Windows 3.1 in enhanced mode with paging enabled (default installation)
- A 386, 486 or Pentium(TM) processor (386sx and 486sx processors are supported)
- 4MB of extended memory is recommended (2MB minimum)
- 1MB of hard disk space for Win32s and another 250K for FreeCell (an optional demonstration program)

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 youve copied the system files from the Win32 SDK. In all cases, the same setup program is used to install and enable Win32s on a Windows 3.1 system.

Making a Floppy-Disk Version of Win32s

The Win32s files are located on the Win32 SDK CD-ROM in the directory \MSTOOLS\ WIN32S\FLOPPY. 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.

1. Place a blank, formatted disk (3.5- or 5.25-inch) in your floppy drive.
2. Copy all files in \mstools\win32s\floppy 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 Disk in your floppy drive.
3. In the File Manager window, 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 the File Manager to access the Win32 SDK CD-ROM.
4. In the File Manager window, display the files in the \mstools\win32s\floppy directory, then double-click on SETUP.EXE to run the Setup program.
5. Follow the setup instructions to complete the installation.

Verifying Win32s Installation

In addition to installing the Win32s system components, the Win32s Setup program optionally installs 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 Applications, which the Win32s Setup program creates.

Note

If installing Win32s on Windows 3.1, it is necessary to run the MS-DOS Share utility before starting Windows. Add SHARE.EXE to your AUTOEXEC.BAT file. SHARE.EXE is not required for Windows for Workgroups.

It is recommended that your CONFIG.SYS file set number of **files** to at least 30. Set **files=30** in CONFIG.SYS if there is no **files** command line or if it specifies less than 30 files.

Disabling Win32s

It should be unnecessary to disable Win32s. The Win32s DLLs will only be loaded when a Win32 application is executed. The Win32s VxD is loaded when Windows starts, but has little memory overhead. If you must disable Win32s or wish to do a clean reinstallation of Win32s, take the following actions:

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

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

- Delete the W32SYS.DLL and the WIN32S16.DLL files from the <windows>\system directory and all files in the <windows>\system\win32s subdirectory (<windows> is the Windows installation directory such as c:\windows.)
- Restart Windows

CHAPTER 2 **Win32s Technical Overview**

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

Features

The Win32s mapping layer allows Win32 applications to make 32-bit calls to Windows 3.1, which is responsible for all graphics and windowing operations. The Windows API in Windows 3.1 and Windows NT have many common features.

The following list highlights Windows 3.1 features available via the Win32s API:

- Complete windowing interfaces (User)
- All graphics functions (GDI)
- OLE (object linking and embedding) version 1.0
- DDE/DDEML (dynamic data exchange and DDE Management Library)
- TrueType fonts
- Common dialogs

In addition to supporting Windows 3.1 functionality, the following Windows NT features are also supported via Win32s on Windows 3.1:

- Structured exception handling (SEH)
- Sparse memory (Virtual memory API)
- Growable heaps (Heap API)
- Named shared memory

The following new features have been added in Win32s 1.1:

- Memory-mapped files (backed by disk image)
- Network support (NetBios and Windows Sockets 1.1∗ APIs)
- Multimedia support (sound APIs)
- International Support (localized versions of Win32s and Code Page/Unicode(TM) translation APIs)

∗ Win32s provides a 32-bit Windows Socket translation layer and requires that your system already has 16-bit Windows Sockets 1.1 installed. There are a number of third party vendors that ship TCP/IP and Windows Sockets products for MS-DOS and Windows.

Win32s offers a 32-bit solution for Windows 3.1 supporting binary compatibility with Windows NT running on 386/486 hardware and source compatibility to RISC platforms such as the R4000 and DEC(TM) Alpha AXP(TM).

Architecture

Figure 1 illustrates the key pieces of Win32s. The dark gray boxes represent Win32s components; the light gray boxes depict Windows 3.1 components. The 32-bit Win32 application dynamically binds to Win32s versions of Win32 system DLLs such as: KERNEL32.DLL, GDI32.DLL, and USER32.DLL. 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 memory management.
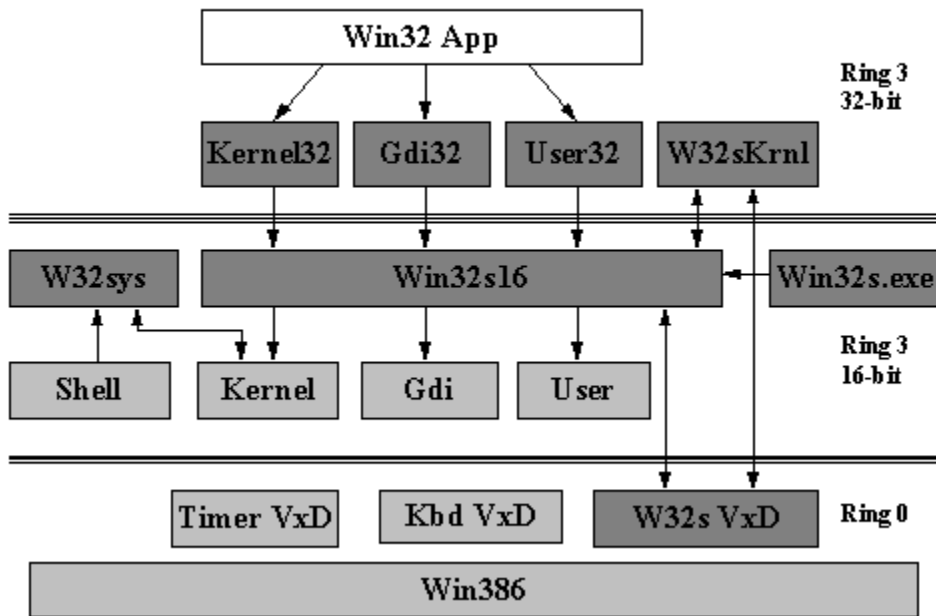

Figure 1. Win32s Architecture.

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

The light gray User, GDI, and Kernel components are the Windows 3.1 window manager, graphics engine, and kernel services DLLs, respectively. Figure 1 shows the general flow of control when a Win32 application makes a Win32 call. Most calls are passed to Windows 3.1, which actually implements the call.

Coordination Between Windows 3.1 and Win32s

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

The first level of coordination occurs in the Windows 3.1 loader. The Windows 3.1 loader was designed to recognize the presence of a Win32s loader. When Windows 3.1 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 Windows NT executable, known as PE for Portable Executable. If the image is not a PE image, Windows 3.1 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 resource loader. One reason to do this is so that

application icons appear correctly in the Program Manager, or other resource viewing utilities. When Windows 3.1 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 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 versioning. Versioning APIs are new to Windows 3.1 and allow version-marked executables to be checked by utilities such as setup tools. Win32s hooks Windows 3.1 so that version calls return valid information when a 16-bit Windows application version-checks a Win32 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 also allows Windows hook APIs and message subclassing to be supported. A Win32 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 and Win32s must also have 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 application and 16-bit Windows system code.

For these reasons, Win32s should be considered an operating system extension to Windows 3.1 that supports 32-bit programming via the Win32 API.

Message Handling

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

Memory

Windows 3.1 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 virtual address space. Win32 applications running on Windows 3.1 coexist in the same shared global memory heap as other Win32 and Win16 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 applications share a single copy of the DLL code and data.
- Win32s DLLs and VxD are shared by all Win32 applications.
- Win32s supports executing data as code, such as dynamic code generation.
- Win32 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 version 1.1 allocates a non-growable stack for each Win32 process up to 128K. Stack size is specified at link time (linker switch); Win32s will create at least a 64K stack.

One impact of a global shared heap and not being able to use selectors for dereferencing addresses (as occurs in Win16 applications) is that 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 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 applications should not be designed to assume that all Win32 applications run in the same address space. Such applications will not run on Windows NT or on future versions of Windows for MS-DOS. Also, Win32s may implement sparse memory using separate address spaces in a future release.

Processes and Tasking

Win16 and Win32 applications coexist seamlessly on the same desktop (display). Win32s assures that Win16 applications see Win32 applications as other Win16 applications, and that Win32 applications see Win16 applications as other Win32 applications. A Win32 application can enumerate windows in the system and will receive 32-bit window handles for both Win32 and Win16 applications. This means that Win32 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 non-preemptive, message-driven scheduler is responsible for scheduling both Win16 and Win32 applications. Win32 applications operate under the same constraints as Win16 applications. To keep the user interface responsive, Win32 applications must check their message queue regularly. On Windows NT, a Win32 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, 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 responsive.

Win32 Dynamic Link Libraries

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

- Win32 DLLs in Windows 3.1 receive the same notifications as on Windows NT 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.

File Input and Output

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 via 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 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 application running with Windows 3.1 and with Windows NT.

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

Printing

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

Win32s supports all Windows 3.1 escapes except those marked as obsolete in the Windows 3.1 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 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. Alternatively, you can call the new Win32 printing API functions that replace the driver-specific escapes when the application runs with Windows NT.

CHAPTER 3 **Win32s Programming Details**

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

Preemptive Multitasking

Windows NT is a preemptive multitasking operating system; Windows 3.1 relies on message-driven multitasking. This difference can cause compatibility problems when Win32 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 applications running with Windows NT. For example, a Win32 application may call a DLL which begins to update DLL-managed data; the preemptive 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** 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 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 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 via a private message in *lParam* is safe between two cooperative Win32 applications; passing a linear 32-bit pointer in *lParam* via a private message to a Win16 application will result in an invalid pointer when the Win16 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 applications. Again, this is 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 applications can intercept messages destined for Win32 applications; and similarly Win32 hooks can intercept messages destined for Win16 applications. In each case the 16- or 32-bit message hook will intercept messages for Win16 and Win32 applications but translated into the appropriate 16- or 32-bit message form.

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 DLLs DEF file:

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

Note:
If your port of a 16-bit Windows 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.

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

On Windows NT:

- **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 block 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.

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

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 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 system-wide index such that a unique TLS index is available for each process which links to a DLL.

In the DLL initialization code fragment below, the DLL will allocate a TLS index for each process that attaches to the DLL on Windows NT. 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.

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

```
BOOL APIENTRY DllEntryPoint(HINSTANCE hinstDll, DWORD fdwReason,
        LPVOID lpvReserved)
{
    static BOOL fFirstProcess = TRUE;
    BOOL    fWin32s = GetVersion() & 0x80000000;
    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 applications that rely on C run-time routines can choose to statically link the C run-time routines with the application or dynamically link to a C run-time DLL. Windows NT provides a C run-time DLL called CRTDLL.DLL located in the SYSTEM32 directory. Applications do not need to ship this C run-time DLL since it is provided by the operating system itself. Win32s also ships a C run-time DLL as part of the licensable Win32s binaries. Therefore, an application can rely on the existence of the C run-time DLL on both Windows NT and Win32s.

Using CRTDLL.DLL is useful for applications that have an executable and multiple DLLs which 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, there are some aspects of CRTDLL.DLL which 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** and **_iob**.

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

The global **_iob** variable is an internal C run-time pointer to stdin/stdout/stderr file handles. If you use CRTDLL.DLL and use standard handles in C run-time calls, such as **fwprintf** (stdout, ...), your application will not load on Win32s and generate a run-time error message about an **_iob** missing export. Use **GetStdHandle** with **ReadFile/WriteFile** or C run-time functions **_open** and **read**/**write**.

If your application consists of a single executable with no DLLs, statically link the C run-time.

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. If an explicit OF_SHARE flag in **OpenFile** is not specified, the default on Windows 3.1 is OF_SHARE_COMPAT. On Windows NT, the default is OF_SHARE_DENY_NONE. OF_SHARE_DENY_NONE does not allow files to be opened with OF_SHARE_COMPAT. **CreateFile** and **fopen** do not support OF_SHARE_COMPAT. A file opened with these functions cannot be open with **OpenFile** or **_lopen** in compatibility mode. For further information, see the **OpenFile** documentation or on-line Win32 API reference.

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

International Support

Win32 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, etc. These error message strings are isolated to the VxD and one 16-bit DLL, W32S.386 and W32SYS.DLL, respectively.

To ship a localized version of a Win32 application with Win32s, you will need to replace the English version of the w32s.386 and w32sys.dll with the appropriate language version of these files from the \mstools\win32s\nls directory according to the table below:

```
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
```

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. See the Win32 SDK Getting Started booklet for CompuServe information.

Code Page and Unicode Translation Support

Windows 3.1 does not support Unicode, so Win32s 1.1 has been implemented without Unicode support. However, Win32 application resources 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 1.1 supports Code Page/Unicode translation via 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 1.1 installs the following OEM and ANSI code pages by default which are appropriate for US and most Western European languages:

```
unicode.nls
c_437.nls
c_850.nls
c_1252.nls
```

The default code pages should be sufficient for most applications, however the following

set of code pages are provided in \mstools\win32s\nls and can be shipped with an application that requires additional code page support; for example an application that wishes to use the Greek code page. The table below lists the code pages supported by the .NLS files provided:

Code Page	Canonical char. set name	ANSI Code Page	OEM Code Page
1250	Windows 3.1 Eastern European		x
1251	Windows 3.1 Cyrillic	x	
1252	Windows 3.1 US (ANSI)	x	
1253	Windows 3.1 Gree	x	
1254	Windows 3.1 Turkish	x	
437	MS-DOS United States		x
850	MS-DOS Multilingual (Latin I)		x
852	MS-DOS Slavic (Latin II)		x
855	IBM Cyrillic (primarily Russian)		
857	IBM Turkish		x
860	MS-DOS Portuguese		x
861	MS-DOS Icelandic		x
863	MS-DOS Canadian-French		x
865	MS-DOS Nordic		x
866	MS-DOS Russian (USSR)		x
869	IBM Modern Greek		

The OEM code page that is installed should match the code page used by MS-DOS which is 437 in the US and generally 850 for Western Europe. Win32s 1.1 installs both of these OEM code pages by default. Your setup program will need to install other OEM code pages to match MS-DOS code pages for other languages.

Note
Use of code pages other than UNICODE.NLS and C_1252.NLS will require 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.

Performance Considerations

Calls to the Win32 API are translated before being 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 functionsfor example, functions that simply query statushave a relatively high thunking overhead. Test scenarios that call a broad selection of Win32 API functions take approximately ten percent longer due to the thunking layer.

Therefore, an application that calls only Win32 API functions will experience a 10% 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 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 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 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 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.
- Local variables (stack variables) are faster to pass by reference between the Win32 application and the system.

Using **PolyLine** saves time in Win32s by minimizing the number of thunks that must be passed through from the Win32 application to Windows 3.1 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 have pointers translated. In general, pointers to local variables will always be translated fastest since they are on the applications stack, and Win32s has optimized translation of the stack pointer.

Advanced Features

Because the entire Win32 API is exported by Win32s, any Win32 application can be loaded into Windows 3.1 since 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 code 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 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 application is running on Windows NT and is *one* if the application is running on Windows 3.1.

An application can selectively, and as a run-time feature, implement different functionality when the application is run with Windows 3.1 via Win32s or when 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. The application can use the error return from these unsupported Win32 API 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, where it runs only as a single-threaded application. But, consider that all Win32 applications automatically benefit from the scheduler on Windows NT even if they do not use multiple threads. Win32 applications execute with an asynchronous messaging model on Windows NT, which allows users to switch away from an applications that are tied up in long calculations. Therefore, taking advantage of threads when an application runs on Windows NT is not required and most applications will just 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 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 application, such as static library code. There is no support for Win32 applications directly loading 16-bit DLLs or vice versa.

Applications that rely on 16-bit DLLs, such as licensed DLLs for which obtaining 32-bit versions is difficult, 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 application via 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 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.

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 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 architect your Win32 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.

To support this application in Windows NT, the DLL should be replaced by a Windows NTspecific 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 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 applications with the *universal thunk* solution.

Universal Thunk

Binary compatibility and the requirement to provide natural migration to Windows NT creates a portability problem for ISVs who have Windows applications that require device drivers. Windows NT does not support mixing 16- and 32-bit code in the same process. The mixing restriction also precludes 16-bit DLLs from being called by Win32 applications.

There are a number of IPC mechanisms, such as DDE, that 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 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 applications.

This design allows a Win32 application to isolate driver-specific routines in a 16-bit DLL. The Win32 application remains portable across Windows 3.1 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.

Universal Thunk Design

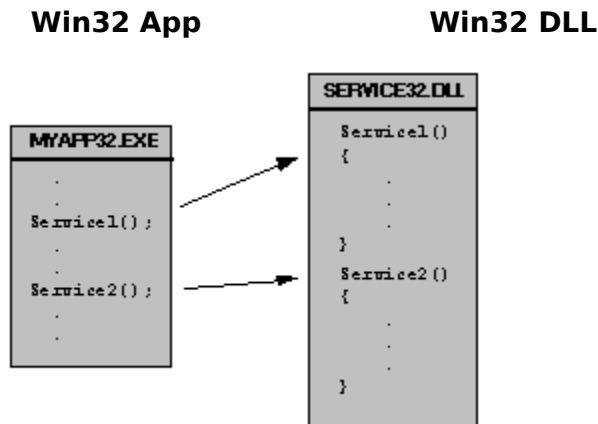Figure 2 illustrates a Win32 application using a Win32 service DLL on Windows NT.

Win32 App   **Win32 DLL**

Figure 2. Win32 application using a Win32-service DLL

Figure 3 illustrates how the same Win32 application running on Windows 3.1 uses the UT mechanism to access services provided by a Win16 DLL. Figure 3 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 and Windows NT. For the service provider, two different modules are required. Both provide the very same services, one using Win32 DLL on Windows NT, and the other using 16-bit DLL by way of UT.

Win32 App Win32 UT DLL   Win16 UT DLL   Win16 DLL

32 BIT   **16 BIT**
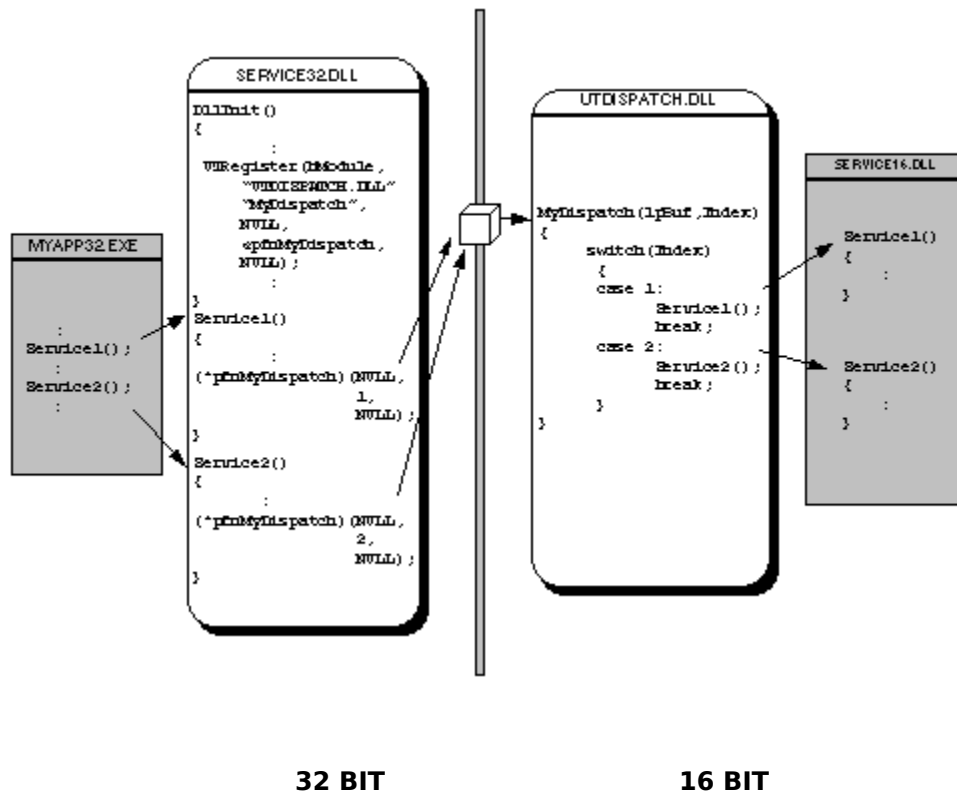
Figure 3. Universal Thunk Design

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 via a 32-bit callable stub from the Win32 application.

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

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 global 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, global memory is used to transfer data.

UTRegister will result in Win32s loading the specified DLL (using **LoadLibrary**) with normal Windows 3.1 DLL initialization occurring. Win32s will then call the **InitFunc** routine (indirectly via 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 via 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. It is the applications responsibility to define the format and packing of the data in the global memory. Care should be taken when passing information in structures via global memory since some data types, such as **int**,

are 32-bits in Win32 and 16-bits in Windows 3.1.

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 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 via 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 Figure 4) which indirectly points to all the pointers to be translated during the thunking process.
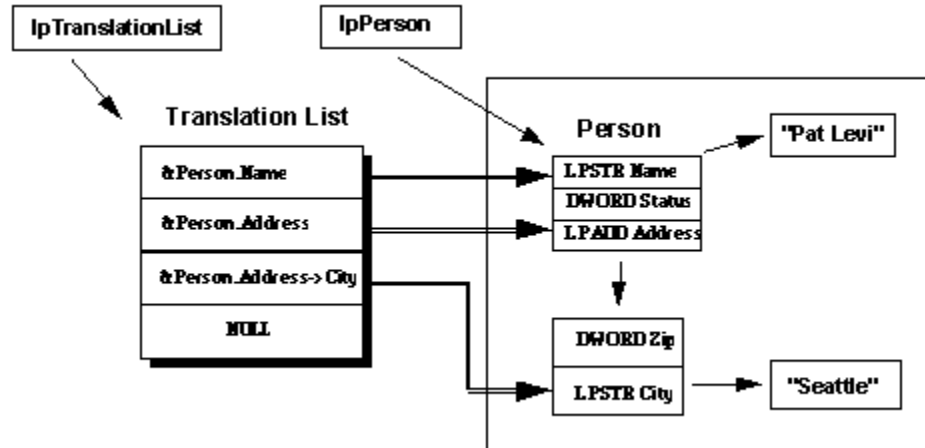
Figure 4. Translation List

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 impact application

usage. The UT APIs, parameters and structures are defined in Appendix C. The UT headers are located in the \MSTOOLS\WIN32S\UT directory.

1. The file W32SUT.H should be included when using UT services. On the 32-bit side W32SUT_32 should be defined before including the file. On the 16-bit side, W32SUT_16 should be defined.

2. A 32-bit DLL using UT services should be linked with the library W32SUT32.LIB. A 16-bit DLL should be linked with the library W32SUT16.LIB.

3. **UTRegister** and **UTUnRegister** are exported by KERNEL32.DLL. **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.

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 via **GlobalAlloc** should be fixed via **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 via **UTSelectorOffsetToLinear** before it can be used by 32-bit code.

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 which 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.

Two samples are provide on the Win32 SDK CD-ROM which 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 APIs and returns data to the Win32 application.

Note

The UT has been tailored to Windows 3.1 which provides a single address space for all processes. Applications which use the UT may have dependencies to Windows 3.1. The UT should be considered a stepping stone to fully 32-bit applications. It is very important to isolate UT code in separate DLLs. These DLLs can be more easily replaced if future operating systems, offering preemptive multitasking and separate address spaces constrain UT support or do not provide a UT at all like Windows NT. The DLL can be replaced with platform-specific versions, isolating the application.

CHAPTER 5 **Win32s Development with the Win32 SDK**

Win32 SDK Support

The Win32 SDK provides all the tools necessary for creating a Win32 application on Windows NT.

After a Win32 application has been debugged and fully tested on Windows NT, you are ready to install Win32s and your Win32 application on Windows 3.1 for final testing and verification. The Win32 SDK provides a floppy-disk image containing a redestributable version of Win32s with a Setup program and FreeCell, a card game test application. You can choose to bundle this disk 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 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 system, first install Win32s by using the Win32s Setup program described in Chapter 1, 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 API function. Win32 functions that cannot

be supported in Windows 3.1 generally errors. A spreadsheet listing all Win32 APIs and whether the API is supported on Win32s 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 on-line help.


Debugging and Testing

The Win32 SDK tools running on Windows NT provides the primary development and debugging environment for creating Win32 applications. A Win32 application should be fully functional and debugged using WinDbg on Windows NT before running the application on Win32s. There are several debugging solutions available to help test and debug Win32 applications running on Windows 3.1: remote WinDbg, debugging DLLs, profiling DLLs, and the Windows 3.1 kernel debugger.

Remote WinDbg

The remote WinDbg solution makes it possible to remotely debug your Win32 application running on Windows 3.1 (a second system) from your Windows NT development machine (primary system). The Windows NT system is the development system with full sources for the application where WinDbg remotely debugs the Win32 binary running on the secondary Windows 3.1 system. Since primary development and debugging should already be complete on Windows NT, this remote debugging technique should only be necessary to debug unique problems that occur when the Win32 application runs on Windows 3.1.
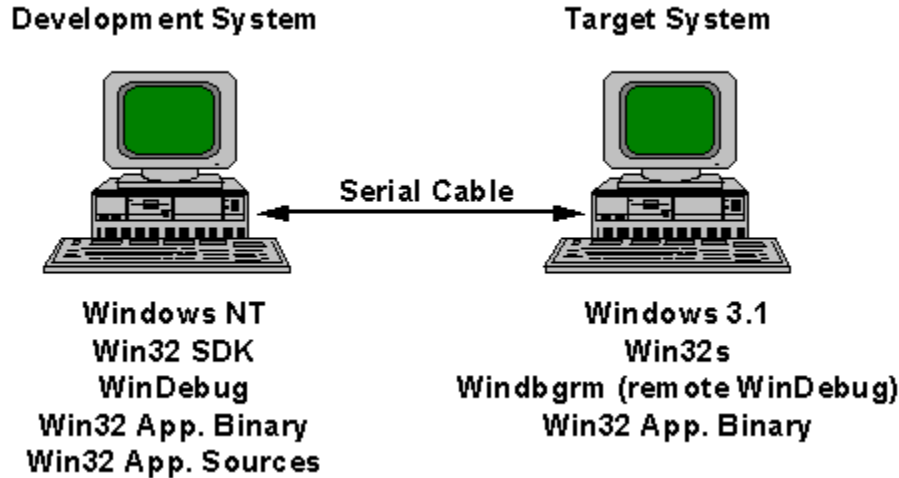
Figure 5. Dual Development/Debugging Systems
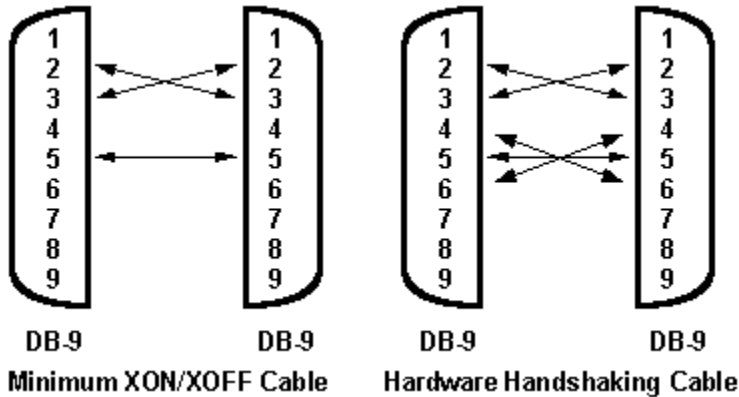
Figure 6. Serial Cable Wiring Diagram

Binary debugging data is passed between the Windows 3.1 and Windows NT systems, therefore 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. Figure 6 illustrates standard cable wiring that should be used with the remote debugger.

Note

The cable layouts in Figure 6 are for standard DB-9 serial connectors. Be sure to refer to your serial card manufacturers 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 setup for HW.

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

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

- WINDBGRM.EXE
- DM32S.DLL
- TLSER32S.DLL

These files are located on the Win32 SDK CD-ROM in the \MSTOOLS\BIN\I386 directory. Figures 7a and 7b illustrate how to setup the WinDbg and WinDbgRM transports to use COM1 at 19,200 baud with XON/XOFF control.
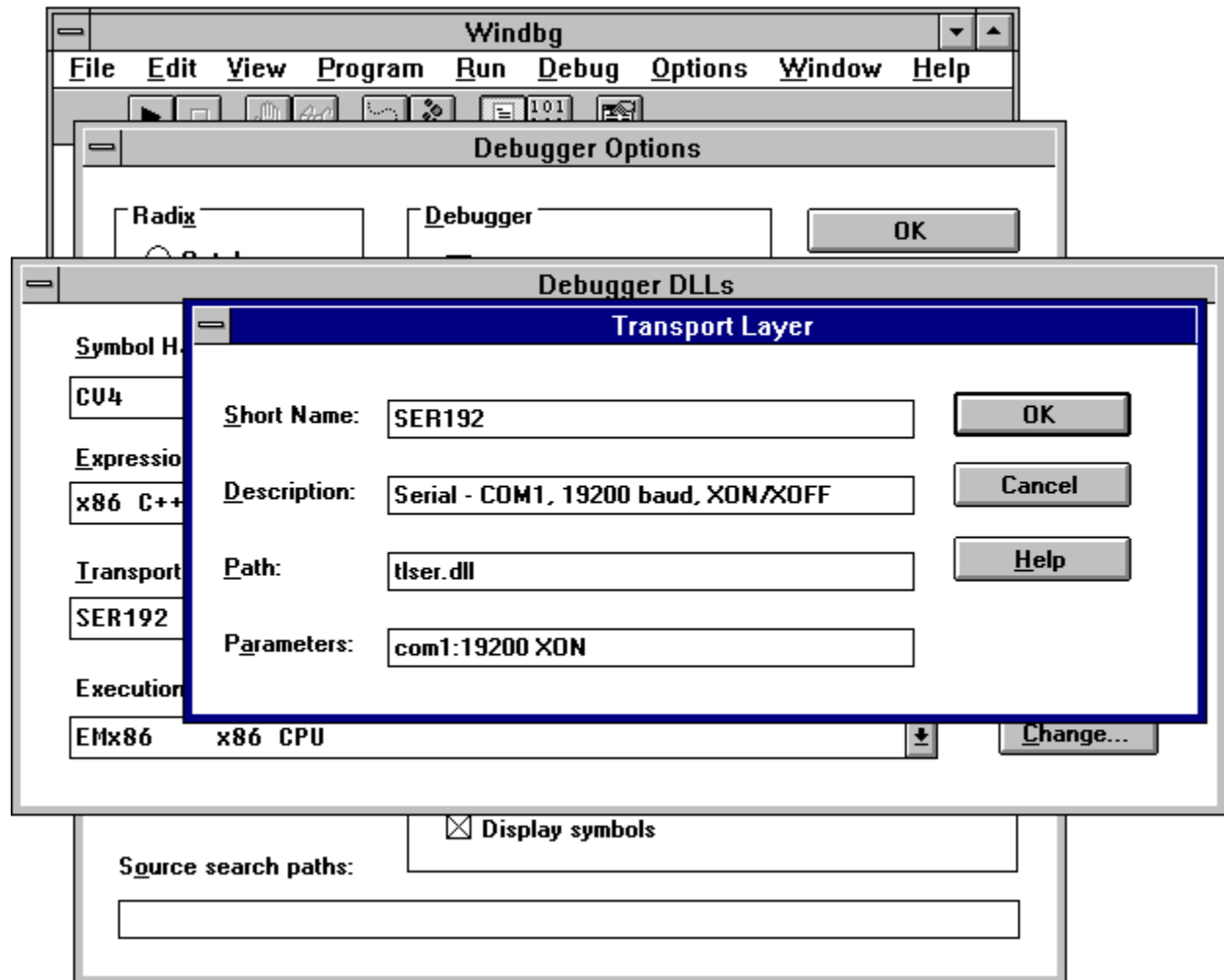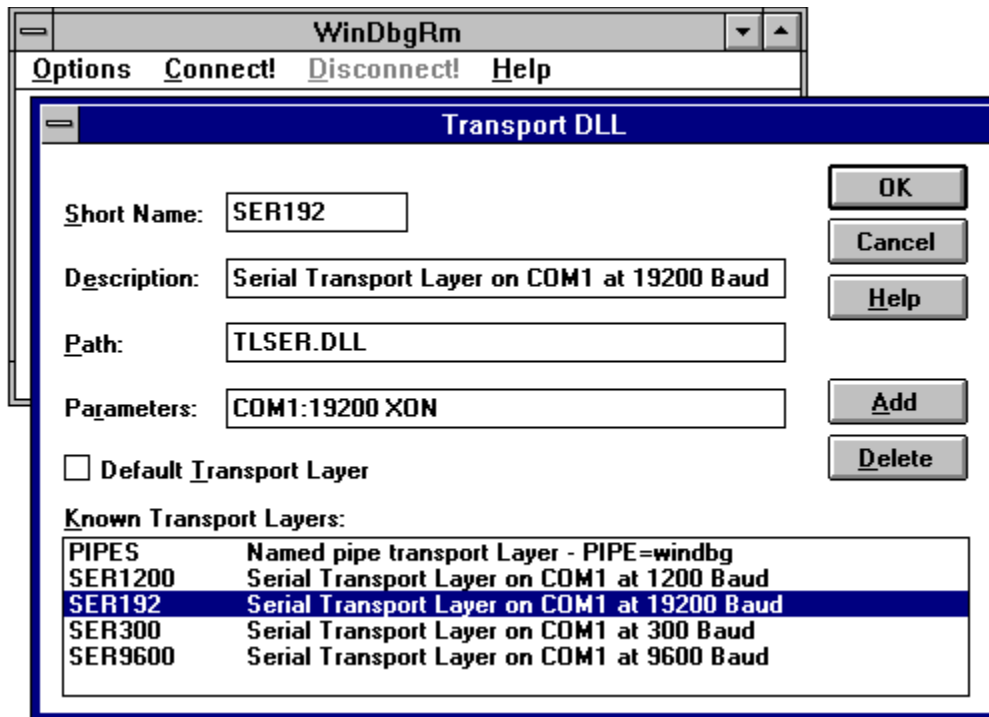

Figure 7a. WinDbg Transport Dialog Box

Figure 7b. WinDbgRm Transport Dialog Box

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 thus saving enormous communication overhead that would otherwise be required if symbols and source code information was 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 directory 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.

Debugging Session Example

To initiate a debugging session, begin with the Windows NT system:

- **windbg** c:\dev\win32app\win32app.exe(where win32app.exe is the name of your binary and c:\dev\win32app is the location to the binary on both systems).
- Select the WinDbg menu Options/Debug Options/Debugger DLLs and set the Transport to SERIAL192. Set the serial port communication parameters.

On the Windows 3.1 system:

- Run windbgrm.exe.
- Select the WinDbgRm menu Options\TransportDLLs and set the Transport to SERIAL192. Adjust the serial port parameters to match the WinDbg on the Windows NT system.
- Press Connect.

Now return to the Windows NT system:

Step into the application. This will establish the communication connection.

- Once a connection is established, WinDbgRm will disappear. When the debugging session ends or the connection is broken, WinDbgRm 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 Windows NT system; you can set breakpoints, single step, display locals, structures, etc.

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 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 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. In addition to the performance overhead present in the debug versions, 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 SDK for the Windows 3.1 debugging DLLs. The Windows 3.1 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 Win32 SDK provides a 32-bit version of Microsoft Test which can be used to test your Win32 applications on Windows NT.

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

The 16-bit Microsoft Test product is available separately from Microsoft and is not part of the Win32 SDK which ships 32-bit tools.

Application Profiling

The Win32 SDK provides tools for profiling Win32 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 applications .EXE and .DLLs (if any) so that it dynamically links to a set of profiling DLLs. This is done separately 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 for Win32s profiling. The profiling DLLs can be placed in the same directory as the Win32 application being profiled.

Once you have verified that profiling is working on Windows NT, you can use the same modified Win32 application for API profiling on Windows 3.1 using the Win32s versions of the profiling DLLs. For more information, see the Win32 SDK *Getting Started*.

Kernel Debugger

The Windows 3.1 kernel debugger is fully documented in the Windows 3.1 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 you have the Windows 3.1 SDK.

Windows NT executables (PE format) support COFF and CV4 debug and symbol information. The Windows 3.1 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). MAPSYMPE.EXE can be run 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 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 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 windows kernel debugger:

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

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

   Create the file win32s.dbg with contents of:

```
/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 thedebugger to stop on abnormal exception 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 break point if the page is currently not present. It is possible to use 386 debug registers (br e address) for up to four break points. This debug flag causes all modules to be pre-loaded so code be disassembled and break-points be 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:
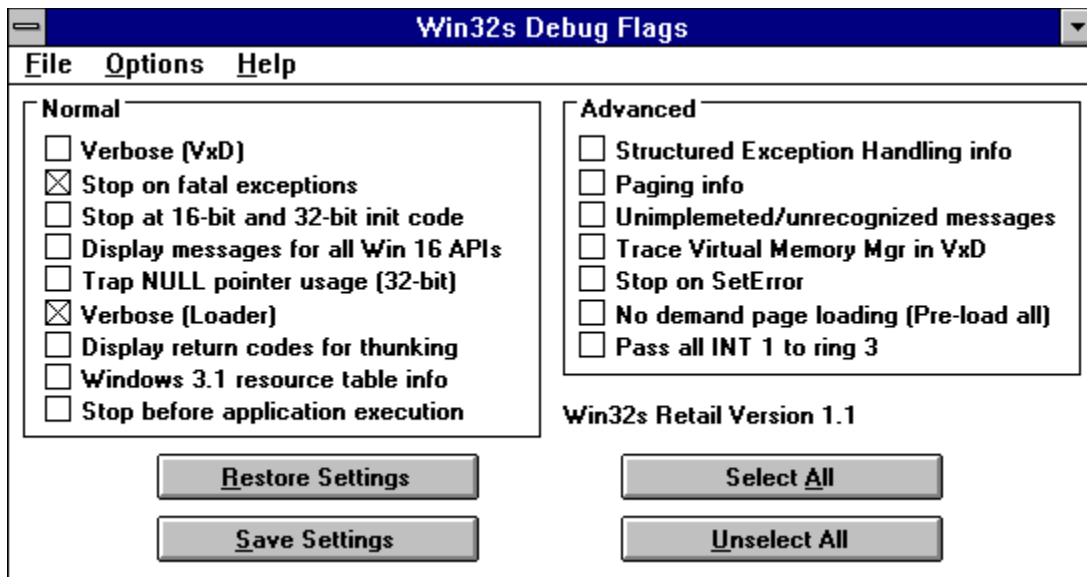
Figure 8. WIN32SDB.EXE Utility

The result of setting the following flags is to make the following change in SYSTEM.INI:

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

5. All Win32 functions are defined in Win32s, so every Win32 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 API function is printed on the debug terminal.
6. Win32s debugging requires 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 click the Change button.
7. You can switch between using 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 system using INSTALL.BAT. Run SWITCH for usage information. This script lets you switch between debug and nondebug versions, allowing complete testing 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.

CHAPTER 6 **Shipping Win32s with Win32 Applications**

Windows 3.1 was released in April of 1992. Therefore, Win32s is not built into Windows 3.1 and it is necessary for software vendors to ship and install Win32s along with the Win32 application in order to work on Windows 3.1. The key feature of Win32s is to allow software developers to ship Win32 applications today for Windows 3.1 and Windows NT that will continue to install and work well on future Windows operating systems. Therefore, it is necessary to create setup programs today that properly handle various setup issues when installing on current and future operating systems.

The Win32s Setup program provided with this Win32 SDK addresses all of the issues raised below. The sources for the Win32s Setup program are also provided. This will allow you to create a custom graphical setup program which addresses the system setup issues correctly and save time by reusing an existing solution. The Win32s Setup program is built using the Setup Toolkit shipped with the Microsoft Windows 3.1 Software Development Kit, Microsoft C 7.0 Compiler, and available separately from Microsoft.

Win32s Installation Rules

In order to ship a single Win32 application which correctly installs on Windows 3.1 and Windows NT, a setup program should meet the following minimum requirements:

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

These minimum installation requirements assure that Win32s files are only installed on systems that require them (Windows 3.1) and not Windows NT since Windows NT is capable of running Win32 applications as-is.

A Win32s setup program should also meet the following criteria:

- Detect Windows 3.0 or lower, provide an informative error message, and do not install Win32s.
- Detect Windows 3.1 running in standard mode, give an informative error message, and do not install Win32s.
- Be prepared for possible future releases of Windows. Win32s should not be installed on any Windows version 4.0 or later since such versions should be able to run Win32 applications directly.
- Detect the presence of Windows NT and only install the Win32 application, not Win32s since Windows NT 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 application).
- Setup should check for 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.

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 isnt 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 APIs 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 (returns 3.1) and past and future versions of Windows.
- **GetWinFlags**: Flag indicates whether Win16 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 RISC-based system. Windows NT runs Win16 applications via an emulator on RISC systems. Therefore, Setup should first determine if the system is Windows NT using **GetWinFlags** (as discussed above), then check the environment variable PROCESSOR_ARCHITECTURE is equal to INTEL before installing an *x*86-based Win32 application (or install the appropriate RISC binary).

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

Before shipping a Win32 application, it is critical to fully test your application on Windows 3.1, Windows for Workgroups, and Windows NT. As discussed previously, the Win32 SDK includes a version of Microsoft Test which can be used to create automated test scripts that test application functionality on both Windows NT and Win32s.

Win32s File Installation and Configuration

This section lists all Win32s files and the directories that the files must reside. The complete set of files must be installed. Partial install will affect other Win32 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. Win32s 1.0 has already been released. Only a Win32s 1.1 set of files must install over an existing 1.0 installation. A Win32s 1.0 setup program must not install over a Win32s 1.1 installation.

The following set of Win32s files must be installed in the <windows>\system directory (where <windows> is the drive and directory that windows has been installed in, such as c:\windows):

```
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.

The following set of Win32s files must be installed in the <windows>\system\win32s directory:

```
advapi32.dll
comdlg32.dll
crtdll.dll
c_1252.nls
c_437.nls
c_850.nls
gdi32.dll
kernel32.dll
lz32.dll
mpr.dll
netapi32.dll
ntdll.dll
olecli32.dll
olesvr32.dll
sck16thk.dll
shell32.dll
unicode.nls
user32.dll
version.dll
w32s.386
w32skrnl.dll
win32s.exe
winmm.dll
winspool.drv
wsock32.dll
```

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

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

Multimedia wave callbacks are enabled by having the WINMM16.DLL loaded when Windows boots. This is accomplished via 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.

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 application and Win32s installation program.

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 is used to determine:
  1. If Win32s is already installed.
  2. Calls **GetWin32sInfo** exported by W32SYS.DLL to determine if installed version of Win32s is earlier release.
  3. Adds the VxD entry to SYSTEM.INI.
- \WIN32S\SETUP\BUILD.BAT is used build the floppy-disk image in \WIN32S\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:

- W32S.LYT contains the names of the files to install and destination directories and whether the files are compressed or not.
- W32SINST.MST contains strings used to add FreeCell to a Program Group named Win32 Applications.
- 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=1.1.0.0
[Nls]
AnsiCP=1252
[FreeCell]
Setup=1
Version=1.1.0.0
```

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 instructs Win32s which code page should be used for translation. If the [Nls] section is not present, code page 1252 is used by default. See the earlier **International Support** section for further information.

The WIN32S.INI file can also be used by Win32 applications to indicate what Win32 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 application entry. The example indicates that Win32s version 1.1 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® C/C++ version 7.0 compiler.

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

APPENDIX A Win32s Version 1.1 Release Notes

Unsupported Features

The following features are being considered for a future release on Win32s 1.1:

- OLE version 2.0
- MAPI
- ODBC
- RPC

The following Win32 features are not currently planned for Win32s:

- Console APIs
- Unicode APIs (Win32s 1.1 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.

APPENDIX B # System Limits

The following programming information lists Windows 3.1 and Windows NT differences that programmers should be aware of when designing a Win32 applications which will run on both platforms. Reviewing these differences can save time developing and debugging applications.

Window Manager (User)

Win32s relies on Windows 3.1 to provide standard dialog controls such as list boxes, combo boxes, and edit controls. Win32s translates messages between the controls and the Win32 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 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 application is run with Windows 3.1.

Controls support specifying limits on the amount of information they will hold, such as EM_SETLIMIT 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 applications local heap is 32-bits and allocated from the global heap, so Windows 3.1 cannot interpret a local handle. Text for multi-line 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 multi-line 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.

When calling the **PeekMessage** function, a Win32 application should not filter any messages for Windows 3.1 private window classes (button, edit, scroll bar, etc.). 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 IDs.

Windows NT dynamically grows the message queue size. Windows 3.1 has a default message queue size of 8 which can be changed by calling **SetMessageQueue**. This call is supported in Win32s and is a NOP on Windows NT.

SetClipboardData must be called with global handles otherwise the data can not 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 applications. However, this value is thunked by Win32s to the Windows 3.1 **EndDialog** function, which will truncate it to 16-bits. Win32s sign-extends the return code from **DialogBox**.

TranslateMessage has additional support on Windows NT than on Windows 3.1. Win32s provides the Windows 3.1 functionality which 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 has a limit of five cached device contexts (DCs). The **GetDC** call obtains a cached DC in Windows 3.1; 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 allows drawing objects (pens and brushes) to be deleted while still selected into a DC. However, the memory allocated for the drawing object remains allocated until the process terminates. Windows NT fails the **DeleteObject** call. Applications should not delete selected objects or they may slowly use up system resources while the application is running on Windows 3.1.

SetDIBitsToDevice is not supported for bitmaps in memory DCs, and is limited to bitmaps of up to 2MB when blting to the display.

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 added additional robustness and process cleanup compared to Windows 3.0, but is not as complete as Windows NT. Win32 applications running on Windows 3.1 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 has a 16-bit world coordinate system. Windows NT supports a 32-bit world coordinate system. Therefore, it is necessary to use the Windows 3.1 limit in developing a Win32 graphics application that will run with both Windows 3.1 and Windows NT.

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

System Services (Kernel)

Win32s processes that are started by Win16 applications should be launched using **WinExec.** The Win16 version of **LoadModule** will not start a Win32s process. If 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.

VirtualAlloc provides a feature on Windows NT allowing theprocess to request the memory allocation at a specified virtual address. This is supported on Win32s but applications should not depend on address range that is available. 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**. 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 **Sleep** Win32 API 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 is a cooperative multitasking system, not a preemptive 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 a file which is opened (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. Therefore, you should be very careful when using **DeleteFile** and must ensure that the file is not in use.

The precision of the time of a file is 2 seconds (this is an MS-DOS limitation).

GetPrivateProfileSection and **GetProfileSection** 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
```

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

Windows 3.1 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 functionality.

Win32 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.

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) which are non-preemptive. Therefore, a portable Win32 WinSocket application should use non-blocking calls and will then run well on Win32s and Windows

NT.

Windows NT supports certain **NetBios** features that are not supported on Win32s due to the non-preemptive shared memory design of Windows 3.1. The ncb_event field of the NCB structure is ignored since Win32s does not implement Windows NT events. Also, Windows NT maintains a per-process name table; Win32s maintains one system-wide name table.

Multimedia

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

Data passed to any Multimedia APIs which is greater than 32K must be allocated via **GlobalAlloc** and locked with **GlobalLock**. Applications must not use **HeapAlloc** or **LocalAlloc** as there is no way to map the data beyond 32K by the Multimedia API.   As mentioned previously, use the GMEM_MOVEABLE flag (not GMEM_FIXED) when using **GlobalAlloc**. GMEM_FIXED will result in locked physical pages in memory.

Win32s API Reference

The following functions are supported by Win32s and define interfaces for Setup programs and the Universal Thunk.

GetWin32sInfo

WORD GetWin32sInfo(*lpInfo*)
LPWIN32SINFO *lpInfo*;

Parameter
lpInfo
    Points to a WIN32SINFO structure, defined as follows:

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

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

1 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

Comments
Function is exported by W32SYS.DLL in Win32s 1.1. 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 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

UTRegister

BOOL UTRegister(*hModule*, *lpsz16BITDLL*, *lpszInitName*, *lpszProcName*, *ppfn32Thunk*, *pfnUT32CallBack*, *lpBuff*)
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.

Returns
Function returns TRUE if the DLL is loaded and **UT16Init** routine was successfully called or FALSE if an error occurred. Use the **GetLastError** function to obtain extended error information. 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).

Comments
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 non-zero value to indicate successful initialization.

The initialization routine must return a non-zero 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
UTUnRegister, **LoadLibrary** (16-bit version), **GetProcAddress** (16-bit version)

UTUnRegister

VOID UTUnRegister(*hModule*)
HANDLE *hModule*

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

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

Returns
No return value.

Comments
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(*pfnUT16CallBack*, *lpBuff*);
UT16CBPROC *pfnUT16CallBack*
LPVOID *lpBuff*

UT16INIT name is a place holder for the application-defined function name. The actual name must be exported by including it in the EXPORTS statement in the DLLs 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.

Returns
1 Upon Success

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

UT16PROC

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

UT16PROC name is a place holder for the application-defined function name. The actual
name must be exported by including it in the EXPORTS statement in the DLLs DEF file.

Parameters
lpBuff
   Pointer to general purpose memory buffer that is passed by the 32-bit code.
dwUserDefined
   Available for application use.
Returns
User defined.

UT16CBPROC

typedef DWORD FAR PASCAL (****UT16CBPROC**)(*lpBuff*, *dwUserDefined*,
**lpTranslationList*);
LPVOID lpBuff
DWORD dwUserDefined
LPVOID *lpTranslationList

Prototype of the 16-bit callable stub which 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 to be 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.

Returns
User defined

See Also
UTRegister

UT32PROC

typedef DWORD (* **WINAPI UT32PROC**)(*lpBuff*, *dwUserDefined*, **lpTranslationList*);
LPVOID *lpBuff*
DWORD *dwUserDefined*
LPVOID **lpTranslationList*

Prototype of the 32-bit callable stub which 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 to be 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.

Returns
User defined.

See Also
UTRegister

UT32CBPROC

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

UT32CBPROC name is a place holder 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.

Returns
User defined

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

See Also
UTRegister

Universal Thunk Auxiliary Services

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

UTSelectorOffsetToLinear

DWORD UTSelectorOffsetToLinear(*lpByte*)
LPVOID *lpByte*;

Parameter
lpByte
  Translate a segmented address to flat form;

Returns
Equivalent flat address.

Comments
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

UTLinearToSelectorOffset

LPVOID UTLinearToSelectorOffset(*lpByte*)
DWORD *lpByte*;

Translate a flat address to a segmented form.

Parameter
lpByte
  Translate a flat address to segmented form.

Returns
Equivalent segmented address (16:16 far pointer).

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

See Also
UTSelectorOffsetToLinear, GlobalFix

Microsoft Win32s Programmer's Reference

Information in this on-line help system is subject to change without notice
and does not represent a commitment on the part of Microsoft Corporation.
The software and/or files described in this on-line help system are furnished
under a license agreement or nondisclosure agreement. The software and/or
files may be used or copied only in accordance with the terms of the
agreement. The purchaser may make one copy of the software for backup
purposes. No part of this on-line help system may be reproduced or
transmitted in any form or by any means, electronic or mechanical, including
photocopying, recording, or information and retrieval systems, for any
purpose other than the purchaser's personal use, without the written
permission of Microsoft Corporation.