Chapter 4

Management Mechanisms

This chapter describes three fundamental mechanisms in Microsoft Windows that are critical to the management and configuration of the system:

- The registry
- Services
- Windows Management Instrumentation

The Registry

The registry plays a key role in the configuration and control of Windows systems. It is the repository for both systemwide and per-user settings. Although most people think of the registry as static data stored on the hard disk, as you'll see in this section, the registry is also a window into various in-memory structures maintained by the Windows executive and kernel. This section isn't meant to be a complete reference to the contents of the Windows registry. That kind of in-depth information is documented in the "Technical Reference to the Windows 2000 Registry" help file in the Windows 2000 resource kits (Regentry.chm), and for Windows XP and Windows Server 2003 that information can be found online as part of the Windows Server 2003 Deployment Kit at *http://www.microsoft.com/windowsserver2003/techinfo/reskit/deploykit.mspx*.

We'll start by providing you with an overview of the registry structure, a discussion of the data types it supports, and a brief tour of the key information Windows maintains in the registry. Then we'll look inside the internals of the *configuration manager*, the executive component responsible for implementing the registry database. Among the topics we'll cover are the internal on-disk structure of the registry, how Windows retrieves configuration information when an application requests it, and what measures are employed to protect this critical system database.

Viewing and Changing the Registry

In general, you should never have to edit the registry directly: application and system settings stored in the registry that might require manual changes should have a corresponding user interface to control their modification. However, as you've already seen a number of times in

this book, some advanced and debug settings have no editing user interface. Therefore, a number of tools are included with Windows that enable you to view and modify the registry.

Windows 2000 comes with two tools for editing the registry—Regedit.exe and Regedt32.exe—whereas Windows XP and Windows Server 2003 have only Regedit.exe. The reason is that the Windows 2000 version of Regedit, which has flexible searching, importing, and exporting capabilities, was ported from Windows 98 and therefore does not support editing or viewing registry security or registry data types not defined on Windows 98. Windows 2000 includes Regedt32 because although it doesn't have as powerful a search feature or support importing and exporting, it was written to run only on Windows 2000 and so it supports security and Windows 2000–specific data types. The Regedit included with Windows XP and Windows Server 2003 includes security editing and knowledge of all registry data types, and thus obviates the need for Regedt32.

There are also a number of command-line registry tools. Reg.exe, for instance, which is included in Windows XP and Windows Server 2003 and available in the Windows 2000 Support Tools, has the ability to import, export, back up, and restore keys, as well as to compare, modify, and delete keys and values.

Registry Usage

There are three principal times that configuration data is read:

- During the boot process, the system reads settings that specify what device drivers to load and how various subsystems—such as the memory manager and process manager—configure themselves and tune system behavior.
- During login, Explorer and other Windows components read per-user preferences from the registry, including network drive-letter mappings, desktop wallpaper, screen saver, menu behavior, and icon placement.
- During their startup, applications read systemwide settings, such as a list of optionally installed components and licensing data, as well as per-user settings that might include menu and toolbar placement and a list of most-recently accessed documents.

However, the registry can be read at other times as well, such as in response to a modification of a registry value or key. Some applications monitor their configuration settings in the registry and read updated settings when they see a change. In general, however, on an idle system there should be no registry activity.

The registry is commonly modified in the following cases:

- Although not a modification, the registry's initial structure and many default settings are defined by a prototype version of the registry that ships on the Windows setup media that is copied onto a new installation.
- Application setup utilities create default application settings and settings that reflect installation configuration choices.

- During the installation of a device driver, the Plug and Play system creates settings in the registry that tell the I/O manager how to start the driver and creates other settings that configure the driver's operation. (See Chapter 9 for more information on how device drivers are installed.)
- When you change application or system settings through user interfaces, the changes are often stored in the registry.

Note Sadly, some applications poll the registry looking for changes when they should be using the registry's *RegNotifyChangeKey* function, which puts a thread to sleep until a change occurs to the area of the registry in which they're interested.

Registry Data Types

The registry is a database whose structure is similar to that of a disk volume. The registry contains *keys*, which are similar to a disk's directories, and *values*, which are comparable to files on a disk. A key is a container that can consist of other keys (*subkeys*) or values. Values, on the other hand, store data. Top-level keys are *root keys*. Throughout this section, we'll use the words *subkey* and *key* interchangeably. (Only root keys are not subkeys.)

Both keys and values borrow their naming convention from the file system. Thus, you can uniquely identify a value with the name mark, which is stored in a key called trade, with the name trade\mark. One exception to this naming scheme is each key's unnamed value. The two Registry Editor utilities, Regedit and Regedt32, display these values differently: Regedit displays the unnamed value as (Default); Regedt32 uses <No Name>.

Values store different kinds of data and can be one of the 15 types listed in Table 4-1. The majority of registry values are REG_DWORD, REG_BINARY, or REG_SZ. Values of type REG_DWORD can store numbers or Booleans (on/off values); REG_BINARY values can store numbers larger than 32 bits or raw data such as encrypted passwords; REG_SZ values store strings (Unicode, of course) that can represent elements such as names, filenames, paths, and types.

Table 4-1 Registry Value Types

Value Type	Description
REG_NONE	No value type.
REG_SZ	Fixed-length Unicode string.
REG_EXPAND_SZ	Variable-length Unicode string that can have embedded environment variables.
REG_BINARY	Arbitrary-length binary data.
REG_DWORD	32-bit number.
REG_DWORD_LITTLE_ENDIAN	32-bit number, with low byte first. This is equivalent to REG_DWORD.
REG_DWORD_BIG_ENDIAN	32-bit number, with high byte first.
REG_LINK	Unicode symbolic link.

Table 4-1 Registry Value Types

Value Type	Description
REG_MULTI_SZ	Array of Unicode NULL-terminated strings.
REG_RESOURCE_LIST	Hardware resource description.
REG_FULL_RESOURCE_DESCRIPTOR	Hardware resource description.
REG_RESOURCE_REQUIREMENTS_LIST	Resource requirements.
REG_QWORD	64-bit number.
REG_QWORD_LITTLE_ENDIAN	64-bit number, with low byte first. This is equivalent to REG_QWORD.
REG_QWORD_BIG_ENDIAN	64-bit number, with high byte first.

The REG_LINK type is particularly interesting because it lets a key transparently point to another key or value. When you traverse the registry through a link, the path searching continues at the target of the link. For example, if \Root1\Link has a REG_LINK value of \Root2\RegKey, and RegKey contains the value RegValue, two paths identify RegValue: \Root1\Link\RegValue and \Root2\RegKey\RegValue. As explained in the next section, Windows prominently uses registry links: three of the six registry root keys are links to subkeys within the three nonlink root keys. Links aren't saved; they must be dynamically created after each reboot.

Registry Logical Structure

You can chart the organization of the registry via the data stored within it. There are six root keys (and you can't add new root keys or delete existing ones) that store information, as shown in Table 4-2.

Table 4-2 The Six Root Keys

Root Key	Description
HKEY_CURRENT_USER	Stores data associated with the currently logged-on user
HKEY_USERS	Stores information about all the accounts on the machine
HKEY_CLASSES_ROOT	Stores file association and Component Object Model (COM) object registration information
HKEY_LOCAL_MACHINE	Stores system-related information
HKEY_PERFORMANCE_DATA	Stores performance information
HKEY_CURRENT_CONFIG	Stores some information about the current hardware profile

Why do root-key names begin with an H? Because the root-key names represent Windows handles (H) to keys (KEY). As mentioned in Chapter 1, HKLM is an abbreviation used for HKEY_LOCAL_MACHINE. Table 4-3 lists all the root keys and their abbreviations. The following sections explain in detail the contents and purpose of each of these six root keys. Again, see the "Technical Reference to the Windows 2000 Registry" help file in the Windows 2000 resource kits or the registry section of the Windows Server 2003 Deployment Kit for details on the contents of these keys.

Table 4-3 Registry Root Keys

Root Key	Abbreviation	Description	Link
HKEY_CURRENT_USER	HKCU	Points to the user profile of the currently logged-on user	Subkey under HKEY_USERS corresponding to currently logged-on user
HKEY_USERS	HKU	Contains subkeys for all loaded user profiles	Not a link
HKEY_CLASSES_ROOT	HKCR	Contains file association and COM registration information	HKLM\SOFTWARE\Classes
HKEY_LOCAL_MACHINE	HKLM	Placeholder—contains other keys	Not a link
HKEY_CURRENT_CONFIG	HKCC	Current hardware profile	HKLM\SYSTEM\CurrentControlSet\Hardware Profiles\Current
HKEY_PERFORMANCE_DATA	HKPD	Performance counters	Not a link

HKEY_CURRENT_USER

The HKCU root key contains data regarding the preferences and software configuration of the locally logged-on user. It points to the currently logged-on user's user profile, located on the hard disk at \Documents and Settings\<username>\Ntuser.dat. (See the section "Registry Internals" later in this chapter to find out how root keys are mapped to files on the hard disk.) Whenever a user profile is loaded (such as at logon time or when a service process runs under the context of a specific username), HKCU is created as a link to the user's key under HKEY_USERS. Table 4-4 lists some of the subkeys under HKCU.

Table 4-4 HKEY_CURRENT_USER Subkeys

Subkey	Description
AppEvents	Sound/event associations
Console	Command window settings (for example, width, height, and colors)
Control Panel	Screen saver, desktop scheme, keyboard, and mouse settings as well as accessibility and regional settings
Environment	Environment variable definitions
Keyboard Layout	Keyboard layout setting (for example, U.S. or U.K.)
Network	Network drive mappings and settings
Printers	Printer connection settings
Software	User-specific software preferences
UNICODE Program Groups	User-specific start menu group definitions
Windows 3.1 Migration Status	File status data for systems that upgrade from Windows 3.x to Windows 2000 and higher

HKEY_USERS

HKU contains a subkey for each loaded user profile and user class registration database on the system. It also contains a subkey named HKU\.DEFAULT that is linked to the profile for the system (which is used by processes running under the local system account and is described in more detail in the section "Services" later in this chapter). This is the profile used by Winlogon, for example, so that changes to the desktop background settings in that profile will be implemented on the logon screen. When a user logs on to a system for the first time and her account does not depend on a roaming domain profile (that is, the user's profile is obtained from a central network location at the direction of a domain controller), the system creates a profile for her account that's based on the profile stored in C:\Documents and Settings\Default User.

The location under which the system stores profiles is defined by the registry value HKLM\Software\Microsoft\Windows NT\CurrentVersion\ProfileList\ProfilesDirectory, which is by default set to %SystemDrive%\Documents and Settings. The ProfileList key also stores the list of profiles present on a system. Information for each profile resides under a subkey that has a name reflecting the Security Identifier (SID) of the account to which the profile corresponds. (See Chapter 8 for more information on SIDs.) Data stored in a profile's key includes the time of the last load of the profile in the *ProfileLoadTimeLow* and *ProfileLoadTimeHigh* values, the binary representation of the account SID in the *Sid* value, and the path to the profile's on-disk hive (which is described later in this chapter in the "Hives" section) in the ProfileImagePath directory. Windows XP and Windows Server 2003 show the list of profiles stored on a system in the User Profiles management dialog box, shown in Figure 4-1, that you access by clicking Settings in the User Profiles section of the Advanced Tab on the System Control Panel applet.

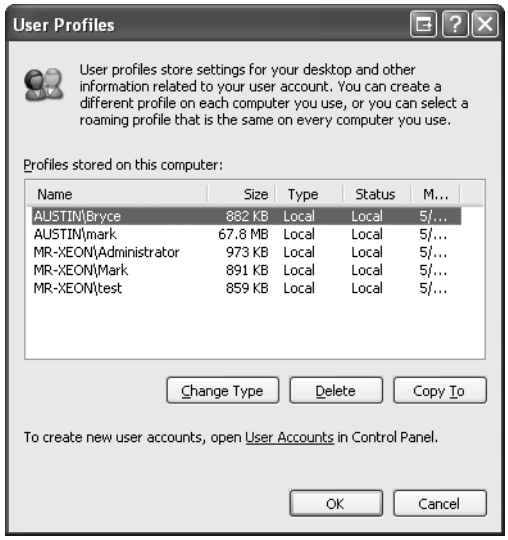


Figure 4-1 The User Profiles management dialog box

EXPERIMENT: Watching Profile Loading and Unloading

You can see a profile load into the registry and then unload by using the Runas command to launch a process in an account that's not currently logged on to the machine. While the new process is running, run Regedit and note the loaded profile key under HKEY_USERS. After terminating the process, perform a refresh in Regedit by pressing the F5 key and the profile should no longer be present.

HKEY_CLASSES_ROOT

HKCR consists of two types of information: file extension associations and COM class registrations. A key exists for every registered filename extension. Most keys contain a REG_SZ value that points to another key in HKCR containing the association information for the class of files that extension represents. For example, HKCR\.xls would point to information on Microsoft Excel files in a key such as HKCU\Excel.Sheet.8. Other keys contain configuration details for COM objects registered on the system.

The data under HKEY_CLASSES_ROOT comes from two sources:

- The per-user class registration data in HKCU\SOFTWARE\Classes (mapped to the file on hard disk \Documents and Settings\<username>\Local Settings\Application Data\Microsoft\Windows\Usrclass.dat)
- Systemwide class registration data in HKLM\SOFTWARE\Classes

The reason that there is a separation of per-user registration data from systemwide registration data is so that roaming profiles can contain these customizations. It also closes a security hole: a nonprivileged user cannot change or delete keys in the systemwide version HKEY_CLASSES_ROOT, and thus cannot affect the operation of applications on the system. Nonprivileged users and applications can read systemwide data and can add new keys and values to systemwide data (which are mirrored in their per-user data), but they can modify existing keys and values in their private data only.

HKEY_LOCAL_MACHINE

HKLM is the root key that contains all the systemwide configuration subkeys: HARDWARE, SAM, SECURITY, SOFTWARE, and SYSTEM.

The HKLM\HARDWARE subkey maintains descriptions of the system's hardware and all hardware device-to-driver mappings. The Device Manager tool (which is available by running System from Control Panel, clicking the Hardware tab, and then clicking Device Manager) lets you view registry hardware information that it obtains by simply reading values out of the HARDWARE key.

EXPERIMENT: Fun with the Hardware Key

You can fool your coworkers or friends into thinking that you have the latest and greatest processor by modifying the value of the *ProcessorNameString* value under HKLM\HARDWARE\DESCRIPTION\System\CentralProcessor\0. The System applet of the control panel displays the *ProcessorNameString* value on the General page. Changes you make to other values in that key, such as the ~MHz, do not have any affect on what the System applet displays, however, because the system caches many of the values for use by functions that applications use to query the system's processor capabilities.

HKLM\SAM holds local account and group information, such as user passwords, group definitions, and domain associations. Windows Server systems that are operating as domain controllers store domain accounts and groups in Active Directory, a database that stores domainwide settings and information. (Active Directory isn't described in this book.) By default, the security descriptor on the SAM key is configured so that even the administrator account doesn't have access.

HKLM\SECURITY stores systemwide security policies and user-rights assignments. HKLM\SAM is linked into the SECURITY subkey under HKLM\SECURITY\SAM. By default, you can't view the contents of HKLM\SECURITY or HKLM\SAM\SAM because the security settings of those keys allow access only by the system account. (System accounts are discussed in greater detail later in this chapter.) You can change the security descriptor to allow read access to administrators, or you can use PsExec to run Regedit in the local system account (as shown in the related experiment for how to do that) if you want to peer inside. However, that glimpse won't be very revealing because the data is undocumented and the passwords are encrypted with one-way mapping—that is, you can't determine a password from its encrypted form.

HKLM\SOFTWARE is where Windows stores systemwide configuration information not needed to boot the system. Also, third-party applications store their systemwide settings here, such as paths to application files and directories, and licensing and expiration date information.

HKLM\SYSTEM contains the systemwide configuration information needed to boot the system, such as which device drivers to load and which services to start. Because this information is critical to starting the system, Windows also maintains a copy of part of this information, called the *last known good control set*, under this key. The maintenance of a copy allows an administrator to select a previously working control set in the case that configuration changes made to the current control set prevent the system from booting. For details on when Windows declares the current control set "good," see the section "Accepting the Boot and Last Known Good."

HKEY_CURRENT_CONFIG

HKEY_CURRENT_CONFIG is just a link to the current hardware profile, stored under HKLM\SYSTEM\CurrentControlSet\Hardware Profiles\Current. Hardware profiles allow

the administrator to configure variations to the base system driver settings. Although the underlying profile might change from boot to boot, applications can always reference the currently active profile through this key. Hardware profile management is managed through the Hardware Profiles dialog box that you access by clicking Settings in the Hardware Profiles section on the Hardware page of the Control Panel's System applet. During the boot process, Ntldr will prompt you to specify which profile it should use if there is more than one.

HKEY_PERFORMANCE_DATA

The registry is the mechanism to access performance counter values on Windows, whether those are from operating system components or server applications. One of the side benefits of providing access to the performance counters via the registry is that remote performance monitoring works "for free" because the registry is easily accessible remotely through the normal registry APIs.

You can access the registry performance counter information directly by opening a special key named HKEY_PERFORMANCE_DATA and querying values beneath it. You won't find this key by looking in the Registry Editor; this key is available only programmatically through the Windows registry functions, such as *RegQueryValueEx*. Performance information isn't actually stored in the registry; the registry functions use this key to locate the information from performance data providers.

You can also access performance counter information by using the Performance Data Helper (PDH) functions available through the Performance Data Helper API (Pdh.dll). Figure 4-2 shows the components involved in accessing performance counter information.

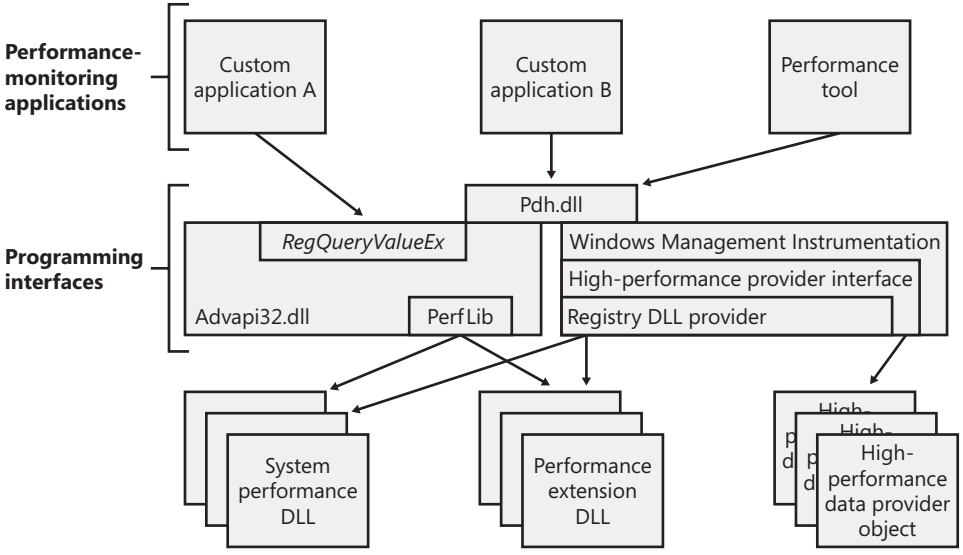


Figure 4-2 Registry performance counter architecture

Troubleshooting Registry Problems

Because the system and applications depend so heavily on configuration settings to guide their behavior, system and application failures can result from changing registry data or security. When the system or an application fails to read settings that it assumes it will always be able to access, it can misbehave by crashing, displaying error messages that hide the root cause, or by not executing with limited functionality. It's virtually impossible to know what registry keys or values are misconfigured without understanding how the system or the application that's failing is accessing the registry. In such situations, the Regmon utility from *www.sysinternals.com* might provide the answer.

Regmon lets you monitor registry activity as it occurs. For each registry access, Regmon shows you the process that performed the access and the time, type, and result of the access. This information is useful for seeing how applications and the system rely on the registry, discovering where applications and the system store configuration settings and troubleshooting problems related to applications having missing registry keys or values. Regmon includes advanced filtering and highlighting so that you can zoom in on activity related to specific keys or values, or to the activity of particular processes.

Regmon Internals

Regmon relies on a device driver that it extracts from its executable image at run time and then starts. Its first execution requires that the account running it have the Load Driver privilege as well as the Debug privilege; subsequent executions in the same boot session require only the Debug privilege because once loaded, the driver remains resident.

There are actually three drivers stored within the Regmon executable: one for use on Windows 95, Windows 98, and Windows Millennium; one for Windows NT, Windows 2000, and Windows XP; and another for use on Windows Server 2003. The reason that there is a driver specific to Windows Server 2003 is that on Windows NT, Windows 2000, and Windows XP the only way for a driver to monitor all registry activity is through system-call hooking and because on Windows Server 2003 a driver can use the *registry callback mechanism* to monitor registry activity. (Windows 95, Windows 98, and Windows Millennium support a different registry monitoring mechanism.)

Recall from the "System Service Dispatching" section of Chapter 3 that system service function addresses are stored in a system service dispatch table in the kernel. A driver can hook a system service by saving the address of a function from the array and replacing the array entry with the address of its hook function. After performing theses steps, any invocations of the hooked system service get diverted to the hooking driver's function, which can examine or modify the parameters to the function and, optionally, execute the original system service function. If it calls the original function, the driver can also examine the result of the operation and examine data the function returns, such as data associated with registry values. Figure 4-3 shows how Regmon intercepts registry functions in kernel mode.

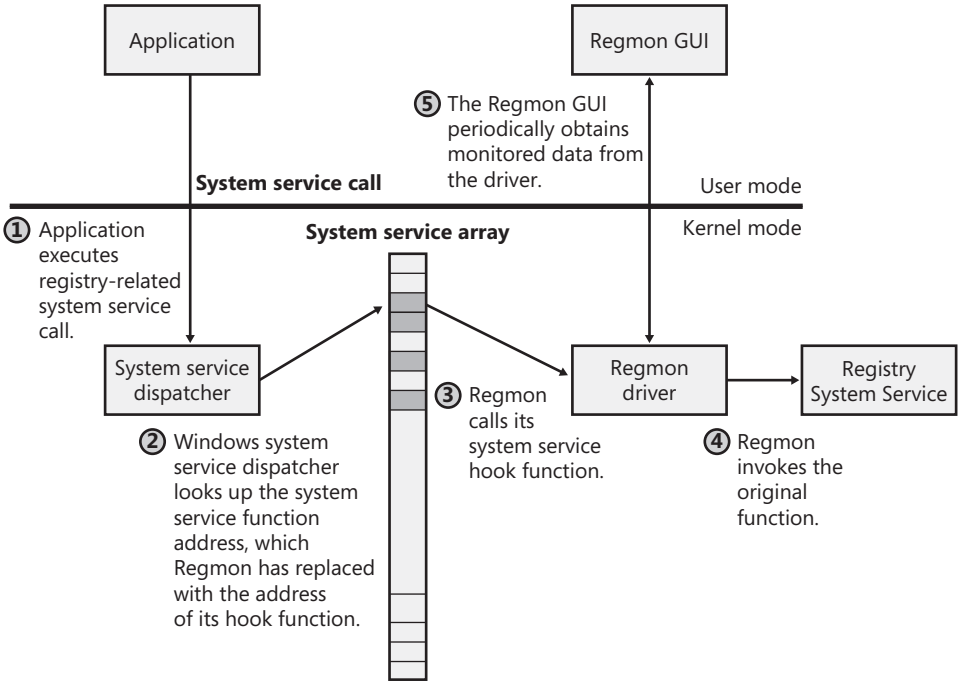


Figure 4-3 Regmon's use of system service hooking

The registry callback mechanism was introduced in Windows XP; however, Regmon still uses system call hooking when run on Windows XP because the callback mechanism on Windows XP does not report all registry activity. When a driver uses the callback mechanism, it registers a callback function with the configuration manager. The configuration manager executes the driver's callback functions at certain points during the execution of registry system services so that the driver has full visibility and control over registry accesses. Antivirus products that scan registry data for viruses or prevent unauthorized processes from modifying the registry are other users of the callback mechanism.

EXPERIMENT: Viewing Registry Activity on an Idle System

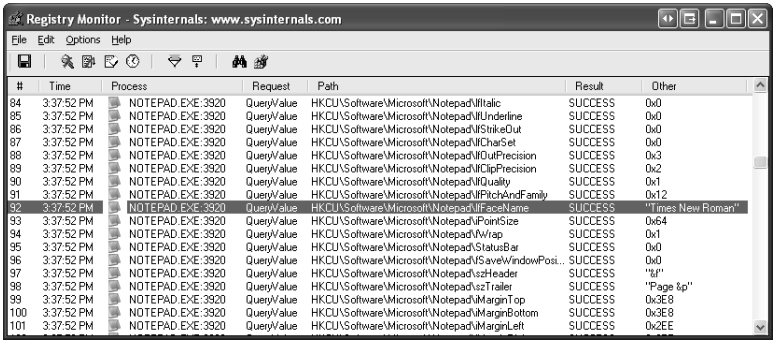
Because the registry implements the *RegNotifyChangeKey* function that applications can use to request notification of registry changes without polling for them, when you Regmon on a system that's idle you should not see repetitive accesses to the same registry keys or values. Any such activity identifies a poorly written application that unnecessarily negatively affects a system's overall performance.

Run Regmon, and after several seconds examine the output log to see whether you can spot polling behavior. Right-click on an output line associated with polling, and choose Process Properties from the context menu to view details about the process performing the activity.

EXPERIMENT: Using Regmon to Locate Application Registry Settings

In some troubleshooting scenarios, you might need to determine where in the registry the system or an application stores particular settings. This experiment has you use Regmon to discover the location of Notepad's settings. Notepad, like most Windows applications, saves user preferences—such as word-wrap mode, font and font size, and window position—across executions. By having Regmon watching when Notepad reads or writes its settings, you can identify the registry key in which the settings are stored. Here are the steps for doing this:

1. Have Notepad save a setting that you can easily search for in a Regmon trace. You can do this by running Notepad, setting the font to Times New Roman, and then exiting Notepad.
2. Run Regmon. Open the highlighting filter dialog box and enter **notepad.exe** in the Include filter. This will have Regmon log only activity that has notepad.exe in either the Process or Path columns.
3. Run Notepad again, and after it has launched stop Regmon's event capture by toggling Capture Events in the Regmon File menu.
4. Scroll to the top line of the resultant log and select it.
5. Press Ctrl+F to open a Find dialog box, and search for **times new**. Regmon should highlight a line like the one shown in the following graphic that represents Notepad reading the font value from the Registry. Other operations in the immediate vicinity should relate to other Notepad settings.



6. Finally, double-click the highlighted line. Regmon will execute Regedit (if it's not already running) and cause it to navigate to and select the Notepad referenced registry value.

Regmon Troubleshooting Techniques

Two basic Regmon troubleshooting techniques are effective for discovering the cause of registry-related application or system problems:

- Look at the last thing in the Regmon trace that the application did before it failed. This action might point to the problem.
- Compare a Regmon trace of the failing application with a trace from a working system.

To follow the first approach, run Regmon and then run the application. At the point the failure occurs, go back to Regmon and stop the logging (by pressing Ctrl+E). Then go to the end of the log and find the last operations performed by the application before it failed (or crashed, hung, or whatever). Starting with the last line, work your way backward, examining the files, registry keys, or both that were referenced—often this will help pinpoint the problem.

Use the second approach when the application fails on one system but works on another. Capture a Regmon trace of the application on the working and failing systems, and save the output to a log file. Then open the good and bad log files with Microsoft Excel (accepting the defaults on the Import wizard), and delete the first three columns. (If you don't delete the first three columns, the comparison will show every line as different because the first three columns contain information that is different from run to run, such as the time and the process ID.) Finally, compare the resulting log files. (You can do this by using WinDiff, which on Windows XP is included in the free support tools on the Windows XP CD, and for Windows 2000 it is included in the Resource Kit.)

Entries in a Regmon trace that have values of "NOTFOUND" or "ACCESS DENIED" in the Result column are ones that you should investigate. NOTFOUND is reported when an application attempts to read from a registry key or value that doesn't exist. In many cases, a missing key or value is innocuous because a process that fails to read a setting from the registry simply falls back on default values. In some cases, however, applications expect to find values for which there is no default and will fail if they are missing.

Access-denied errors are a common source of registry-related application failures and occur when an application doesn't have permission to access a key the way that it wants. Applications that do not validate registry operation results or perform proper error recovery will fail.

A common result string that might appear suspicious is BUFROVERFLOW. It does not indicate a buffer-overflow exploit in the application that receives it. Instead, it's used by the configuration manager to inform an application that the buffer it specified to store a registry value is too small to hold the value. Application developers often take advantage of this behavior to determine how large a buffer to allocate to store a value. They first perform a registry query with a 0-length buffer that returns a buffer-overflow error and the length of the data it attempted to read. The application then allocates a buffer of the indicated size and rereads the value. You should therefore see operations that return BUFROVERFLOW repeat with a successful result.

In one example of Regmon being used to troubleshoot a real problem, it saved a user from doing a complete reinstall of his Windows XP system. The symptom was that Internet Explorer would hang on startup if the user did not first manually dial the Internet connection. This Internet connection was set as the default connection for the system, so starting Internet Explorer should have caused an automatic dial-up to the Internet (because Internet Explorer was set to display a default home page upon startup).

An examination of a Regmon log of Internet Explorer startup activity, going backward from the point in the log where Internet Explorer hung, showed a query to a key under HKCU\Software\Microsoft\RAS Phonebook. The user reported that he had previously uninstalled the dialer program associated with the key and manually created the dial-up connection. Because the dial-up connection name did not match that of the uninstalled dialer program, it appeared that the key had not been deleted by the dialer's uninstall program and that it was causing Internet Explorer to hang. After the key was deleted, Internet Explorer functioned as expected.

Logging Activity in Unprivileged Accounts or During Logon/Logoff

A common application-failure scenario is that an application works when run in an account that has Administrative group membership but not when run in the account of an unprivileged user. As described earlier, executing Regmon requires security privileges that are not normally assigned to standard user accounts, but you can capture a trace of applications executing in the logon session of an unprivileged user by using the Runas command to execute Regmon in an administrative account.

If a registry problem relates to account logon or logoff, you'll also have to take special steps to be able to use Regmon to capture a trace of those phases of a logon session. Applications that are run in the local system account are not terminated when a user logs off, and you can take advantage of that fact to have Regmon run through a logoff and subsequent logon. You can launch Regmon in the local system account either by using the *At* command that's built into Windows and specifying the */interactive* flag, or by using the PsExec utility from *www.sysinternals.com*, like this:

psexec −i −s −d c:\regmon.exe

The -i switch directs PsExec to have Regmon's window appear on the interactive console, the -s switch has PsExec run Regmon in the local system account, and the -d switch has PsExec launch Regmon and exit without waiting for Regmon to terminate. When you execute this command, the instance of Regmon that executes will survive logoff and reappear on the desktop when you log back on, having captured the registry activity of both actions.

Another way to monitor registry activity during the logon, logoff, boot, or shut down process is to use the Regmon log boot feature, which you can enable by selecting Log Boot in the Options menu. The next time you boot the system, the Regmon device driver logs registry activity from early in the boot to the \Windows\Regmon.log. It will continue logging to that file until disk space runs out, the system shuts down, or you run Regmon. A log file storing a registry trace of startup, logon, logoff, and shut down on a Windows XP system will typically be between 50 and 150 MB in size.

Registry Internals

In this section, you'll find out how the configuration manager—the executive subsystem that implements the registry—organizes the registry's on-disk files. We'll examine how the configuration manager manages the registry as applications and other operating system components read and change registry keys and values. We'll also discuss the mechanisms by which the configuration manager tries to ensure that the registry is always in a recoverable state, even if the system crashes while the registry is being modified.

Hives

On disk, the registry isn't simply one large file but rather a set of discrete files called *hives*. Each hive contains a registry tree, which has a key that serves as the root or starting point of the tree. Subkeys and their values reside beneath the root. You might think that the root keys displayed by the Registry Editor tools correlate to the root keys in the hives, but such is not the case. Table 4-5 lists registry hives and their on-disk filenames. The pathnames of all hives except for user profiles are coded into the configuration manager. As the configuration manager loads hives, including system profiles, it notes each hive's path in the values under the HKLM\SYSTEM\CurrentControlSet\Control\hivelist subkey, removing the path if the hive is unloaded. (User profiles are unloaded when not referenced.) It creates the root keys, linking these hives together to build the registry structure you're familiar with and that the Registry Editor displays.

Table 4-5 On-Disk Files Corresponding to Paths in the Registry

Hive Registry Path	Hive File Path
HKEY_LOCAL_MACHINE\SYSTEM	\Windows\System32\Config\System
HKEY_LOCAL_MACHINE\SAM	\Windows\System32\Config\Sam
HKEY_LOCAL_MACHINE\SECURITY	\Windows\System32\Config\Security
HKEY_LOCAL_MACHINE\SOFTWARE	\Windows\System32\Config\Software
HKEY_LOCAL_MACHINE\HARDWARE	Volatile hive
HKEY_LOCAL_MACHINE\SYSTEM\Clone	Volatile hive (on Windows 2000 only)
HKEY_USERS\<security ID of username>	\Documents and Settings\<username>\Ntuser.dat
HKEY_USERS\<security ID of username>_Classes	\Documents and Settings\<username>\Local Settings\Application Data\Microsoft\Windows\Usrclass.dat
HKEY_USERS\.DEFAULT	\Windows\System32\Config\Default

You'll notice that some of the hives listed in Table 4-5 are volatile and don't have associated files. The system creates and manages these hives entirely in memory; the hives are therefore temporary. The system creates volatile hives every time it boots. An example of a volatile hive is the HKLM\HARDWARE hive, which stores information about physical devices and the devices' assigned resources. Resource assignment and hardware detection occur every time the system boots, so not storing this data on disk is logical.

EXPERIMENT: Manually Loading and Unloading Hives

Regedt32 on Windows 2000 and Regedit on Windows XP and Windows Server 2003 have the ability to load hives that you can access through its File menu. This capability can be useful in troubleshooting scenarios where you want to view or edit a hive from an unbootable system or a backup medium. In this experiment, you'll use Regedt32 (if you're running Windows 2000) or Regedit (if you're running Windows XP and Windows Server 2003) to load a version of the HKLM\SYSTEM hive that Windows Setup creates and stores in \Windows\Repair during the install process.

1. Hives can be loaded only underneath HKLM or HKU, so open Regedit or Regedt32, select HKLM, and choose Load Hive from the Regedit File menu or the Regedt32 Registry menu.
2. Navigate to the \Windows\Repair directory in the Load Hive dialog box, select System.bak, and open it. When prompted, enter **Test** as the name of the key under which it will load.
3. Open the newly created HKLM\Test key, and explore the contents of the hive.
4. Open HKLM\System\CurrentControlSet\Control\Hivelist, and locate the entry \Registry\Machine\Test, which demonstrates how the configuration manager lists loaded hives in the HiveList key.
5. Select HKLM\Test, and choose Unload Hive from the Regedit File menu or the Regedt32 Registry menu to unload the hive.

Hive Size Limits

In some cases, hive sizes are limited. For example, Windows places a limit on the size of the HKLM\SYSTEM hive. It does so because Ntldr reads the entire HKLM\SYSTEM hive into physical memory near the start of the boot process when virtual memory paging is not enabled. Ntldr also loads Ntoskrnl and boot device drivers into physical memory, so it must constrain the amount of physical memory assigned to HKLM\SYSTEM. (See Chapter 6 for more information on the role Ntldr plays during the startup process.) On Windows 2000, Ntldr places a fixed upper limit on its size of 12 MB, but on Windows XP and Windows Server 2003 it is more flexible, allowing the hive to be up to 200 MB or one fourth the amount of physical memory on the system, whichever is lower.

On Windows 2000, there is also a limit on the combined sizes of all loaded registry hives. Windows 2000 uses a type of kernel memory called *paged pool* to hold registry hives in memory, and therefore, the total amount of loaded registry data is constrained by the amount of paged pool that's available. The amount of paged pool the memory manager creates during its initialization is based on a number of factors, such as the amount of physical memory on the system. On a system where the memory manager creates the largest amount of paged pool

possible, the registry size limit is 376 MB. Because a system will not operate smoothly if there is not enough paged pool left over for other uses, Windows 2000 won't let registry data grow to more than 80 percent of paged pool and also honors a user-configurable registry quota if it's less than that amount. Click the Change button in the Virtual Memory section of the Performance Options dialog box that you reach on the Advanced page of the Control Panel's System applet to view or modify the registry quota setting, which you can see in Figure 4-4.

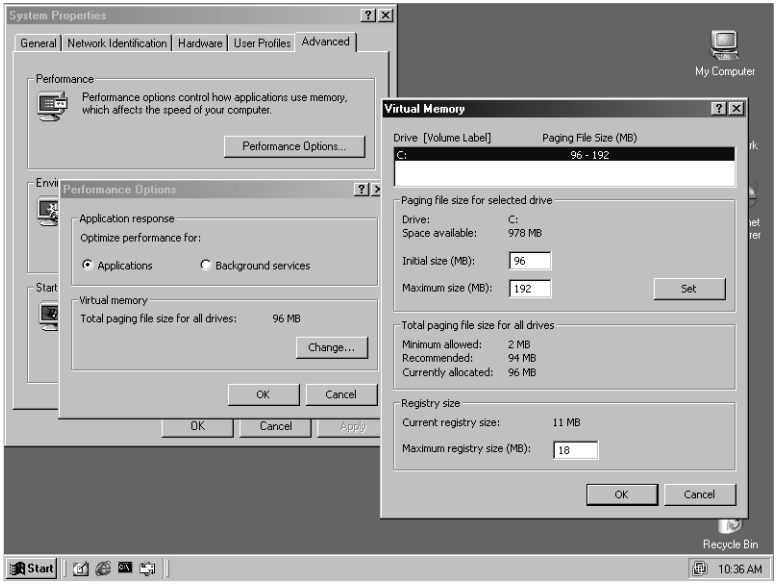


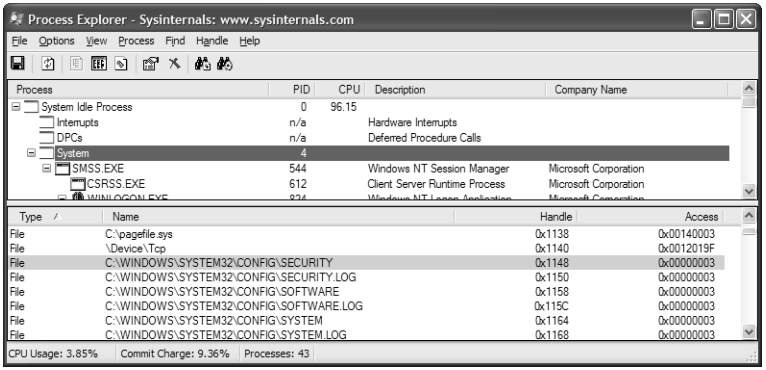
Figure 4-4 Windows 2000 registry quota setting

The upper limit on the total size of loaded registry hives can create a limit on the number of concurrently logged-in users on a Windows 2000 system running Terminal Services, because each user's profile contributes to the loaded hive size. On Windows XP and Windows Server 2003, the configuration manager therefore does not use paged pool and instead relies on the memory manager's memory-mapping functions to map into system memory only the portions of registry hives that it's accessing at any given point in time. There is no registry quota on Windows XP or Windows Server 2003, and the total size of loaded hives does not constrain the scalability of Terminal Services.

EXPERIMENT: Looking at Hive Handles

The configuration manager opens hives by using the kernel handle table (described in Chapter 3) so that it can access hives from any process context. Using the kernel handle table is an efficient alternative to approaches that involve using drivers or executive components to access from the system process only handles that must be protected from user processes. You can use the Process Explorer utility, available from *www.sysinternals.com*, to see the hive handles. On Windows 2000, the object manager reports kernel

handle table handles as being opened in the System Idle process, and on Windows XP and Windows Server 2003 it reports them as being opened in the System process. Select the appropriate process for the Windows version that you are running, and select Handles from the Lower Pane View menu entry in the View menu. Sort by handle type, and scroll until you see the hive files, as shown in the following graphic.



A special type of key known as a *symbolic link* makes it possible for the configuration manager to link hives to organize the registry. A symbolic link is a key that redirects the configuration manager to another key. Thus, the key HKLM\SAM is a symbolic link to the key at the root of the SAM hive.

Hive Structure

The configuration manager logically divides a hive into allocation units called *blocks* in much the same way that a file system divides a disk into clusters. By definition, the registry block size is 4096 bytes (4 KB). When new data expands a hive, the hive always expands in block-granular increments. The first block of a hive is the *base block*. The base block includes global information about the hive, including a signature—*regf*—that identifies the file as a hive, updated sequence numbers, a time stamp that shows the last time a write operation was initiated on the hive, the hive format version number, a checksum, and the hive file's internal file-name (for example, \Device\HarddiskVolume1\WINDOWS\SYSTEM32\CONFIG\SAM). We'll clarify the significance of the updated sequence numbers and time stamp when we describe how data is written to a hive file. The hive format version number specifies the data format within the hive. The configuration manager uses hive format version 1.3 on Windows 2000. On Windows XP and Windows Server 2003, it uses format version 1.3 for all hives except for System and Software for roaming profile compatibility with Windows 2000. For System and Software hives, it uses version 1.5 because of the new format's optimizations for large values and searching.

Windows organizes the registry data that a hive stores in containers called *cells*. A cell can hold a key, a value, a security descriptor, a list of subkeys, or a list of key values. A field at the

beginning of a cell's data describes the data's type. Table 4-6 describes each cell data type in detail. A cell's header is a field that specifies the cell's size. When a cell joins a hive and the hive must expand to contain the cell, the system creates an allocation unit called a *bin*. A bin is the size of the new cell rounded up to the next block boundary. The system considers any space between the end of the cell and the end of the bin to be free space that it can allocate to other cells. Bins also have headers that contain a signature, *hbin*, and a field that records the offset into the hive file of the bin and the bin's size.

Table 4-6 Cell Data Types

Data Type	Description
Key cell	A cell that contains a registry key, also called a *key node*. A key cell contains a signature (*kn* for a key, *kl* for a symbolic link), the time stamp of the most recent update to the key, the cell index of the key's parent key cell, the cell index of the subkey-list cell that identifies the key's subkeys, a cell index for the key's security descriptor cell, a cell index for a string key that specifies the class name of the key, and the name of the key (for example, CurrentControlSet).
Value cell	A cell that contains information about a key's value. This cell includes a signature (*kv*), the value's type (for example, REG_ DWORD or REG_BINARY), and the value's name (for example, Boot-Execute). A value cell also contains the cell index of the cell that contains the value's data.
Subkey-list cell	A cell composed of a list of cell indexes for key cells that are all subkeys of a common parent key.
Value-list cell	A cell composed of a list of cell indexes for value cells that are all values of a common parent key.
Security-descriptor cell	A cell that contains a security descriptor. Security-descriptor cells include a signature (ks) at the head of the cell and a reference count that records the number of key nodes that share the security descriptor. Multiple key cells can share security-descriptor cells.

By using bins, instead of cells, to track active parts of the registry, Windows minimizes some management chores. For example, the system usually allocates and deallocates bins less frequently than it does cells, which lets the configuration manager manage memory more efficiently. When the configuration manager reads a registry hive into memory, it can choose to read only bins that contain cells (that is, active bins) and to ignore empty bins. When the system adds and deletes cells in a hive, the hive can contain empty bins interspersed with active bins. This situation is similar to disk fragmentation, which occurs when the system creates and deletes files on the disk. When a bin becomes empty, the configuration manager joins to the empty bin any adjacent empty bins to form as large a contiguous empty bin as possible. The configuration manager also joins adjacent deleted cells to form larger free cells. (The configuration manager shrinks a hive only when bins at the end of the hive become free. You can compact the registry by backing it up and restoring it using the Windows *RegSaveKey* and *RegReplaceKey* functions, which are used by the Windows Backup utility.)

The links that create the structure of a hive are called *cell indexes*. A cell index is the offset of a cell into the hive file. Thus, a cell index is like a pointer from one cell to another cell that the configuration manager interprets relative to the start of a hive. For example, as you saw in Table 4-6, a cell that describes a key contains a field specifying the cell index of its parent key; a cell index for a subkey specifies the cell that describes the subkeys that are subordinate to the specified subkey. A subkey-list cell contains a list of cell indexes that refer to the subkey's key cells. Therefore, if you want to locate, for example, the key cell of subkey A, whose parent is key B, you must first locate the cell containing key B's subkey list using the subkey-list cell index in key B's cell. Then you locate each of key B's subkey cells by using the list of cell indexes in the subkey-list cell. For each subkey cell, you check to see whether the subkey's name, which a key cell stores, matches the one you want to locate, in this case, subkey A.

The distinction between cells, bins, and blocks can be confusing, so let's look at an example of a simple registry hive layout to help clarify the differences. The sample registry hive file in Figure 4-5 contains a base block and two bins. The first bin is empty, and the second bin contains several cells. Logically, the hive has only two keys: the root key Root, and a subkey of Root, Sub Key. Root has two values, *Val 1* and *Val 2*. A subkey-list cell locates the root key's subkey, and a value-list cell locates the root key's values. The free spaces in the second bin are empty cells. Figure 4-5 doesn't show the security cells for the two keys, which would be present in a hive.

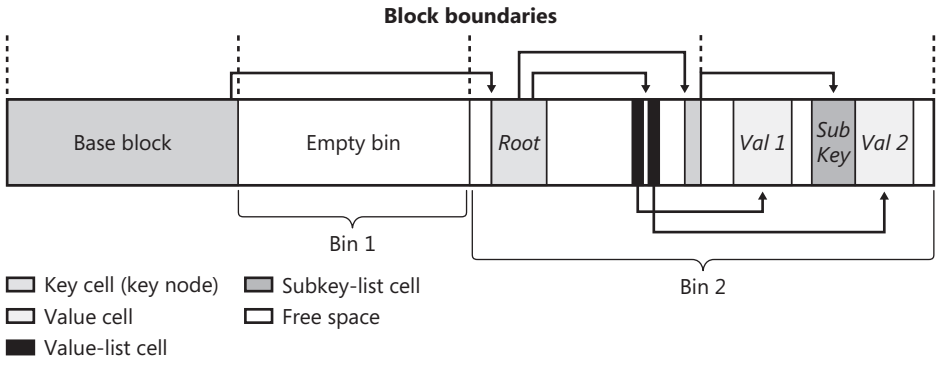


Figure 4-5 Internal structure of a registry hive

Figure 4-6 shows an example of the Disk Probe utility (Dskprobe.exe) examining the first bin in a SYSTEM hive. Notice the bin's signature, *hbin*, at the top right side of the image. Look beneath the bin signature and you'll see the signature *nk*. This signature is the signature of a key cell (*kn*). The signature displays backward because of the way x86 computers store data. The cell is the SYSTEM hive's root cell, which the configuration manager has named internally $$$PROTO.HIV, as specified by the name that follows the *nk* signature.

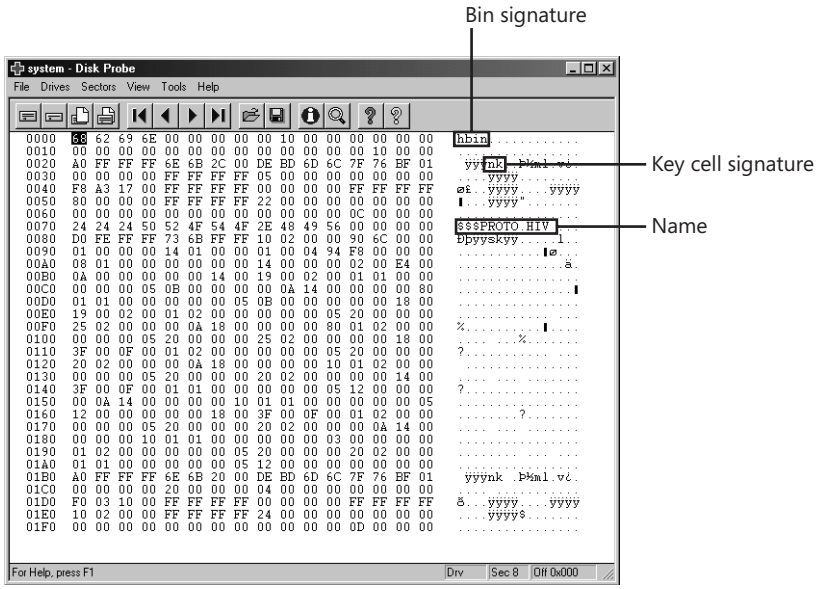


Figure 4-6 Binary contents of first bin in the SYSTEM hive

To optimize searches for both values and subkeys, the configuration manager sorts subkey-list cells alphabetically. The configuration manager can then perform a binary search when it looks for a subkey within a list of subkeys. The configuration manager examines the subkey in the middle of the list, and if the name of the subkey the configuration manager is looking for is alphabetically before the name of the middle subkey, the configuration manager knows that the subkey is in the first half of the subkey list; otherwise, the subkey is in the second half of the subkey list. This splitting process continues until the configuration manager locates the subkey or finds no match. Value-list cells aren't sorted, however, so new values are always added to the end of the list.

Cell Maps

The configuration manager doesn't access a hive's image on disk every time a registry access occurs. Windows 2000 keeps a version of every hive in the kernel's address space. When a hive initializes, the configuration manager determines the size of the hive file, allocates enough memory from the kernel's paged pool to store it, and reads the hive file into memory. (For more information on paged pool, see Chapter 7.) Because all loaded registry hives are read into paged pool, that registry data is typically the largest consumer of the paged pool in Windows 2000. (To check paged pool allocation, use the Poolmon utility, described in the "Experiment: Monitoring Pool Usage" sidebar in Chapter 7.)

In Windows XP and Windows Server 2003, the configuration manager maps portions of a hive into memory as it needs to access them. It uses the cache manager's file mapping functions to map in 16-KB views into the hive files. (See Chapter 10 for more information on the cache manager.) To prevent hive mapping from consuming all the cache manager's address range, the

configuration manager tries to keep no more than 256 views of a hive mapped at any given point in time by unmapping least-recently used (LRU) views when it reaches that limit. The configuration manager still uses the paged pool to store various data structures (including the LRU list of views), but its use of the paged pool is a fraction of what it is in Windows 2000.

Note On Windows XP and Windows Server 2003, the configuration manager will store a block in the paged pool instead of mapping it if the block exceeds 256 KB in size.

If hives never grew, the configuration manager could perform all its registry management on the in-memory version of a hive as if the hive were a file. Given a cell index, the configuration manager could calculate the location in memory of a cell simply by adding the cell index, which is a hive file offset, to the base of the in-memory hive image. Early in the system boot, this process is exactly what Ntldr does with the SYSTEM hive: Ntldr reads the entire SYSTEM hive into memory as a read-only hive and adds the cell indexes to the base of the in-memory hive image to locate cells. Unfortunately, hives grow as they take on new keys and values, which means the system must allocate paged pool memory to store the new bins that contain added keys and values. Thus, the paged pool that keeps the registry data in memory isn't necessarily contiguous.

EXPERIMENT: Viewing Hive Paged Pool Usage

There are no administrative-level tools that show you the amount of paged pool that registry hives, including user profiles, are consuming on Windows 2000. However, the *!reg dumppool* kernel debugger command shows you not only how many pages of the paged pool each loaded hive consumes but also how many of the pages store volatile and nonvolatile data. The command prints the total hive memory usage at the end of the output. (The command shows only the last 32 characters of a hive's name.)

```
kd> !reg dumppool

dumping hive at e20d66a8 (a\Microsoft\Windows\UsrClass.dat)
  Stable Length = 1000
  1/1 pages present
  Volatile Length = 0

dumping hive at e215ee88 (ettings\Administrator\ntuser.dat)
  Stable Length = f2000
  242/242 pages present
  Volatile Length = 2000
  2/2 pages present

dumping hive at e13fa188 (\SystemRoot\System32\Config\SAM)
  Stable Length = 5000
  5/5 pages present
  Volatile Length = 0

...
```

EXPERIMENT: Viewing Hive Memory Usage

In Windows XP and Windows Server 2003, you can view statistics on hive memory usage, including its stable (on-disk) size and nonvolatile size, the number of active views, and the number of views that are locked into memory, using the *!reg hivelist* command (note that the line output wraps):

```
-------------------------------------------------------------------------------------
----------------------
| HiveAddr |Stable Length|Stable Map|Volatile Length|Volatile Map|MappedViews|PinnedVi
ews|U(Cnt)| BaseBlock | FileName
-------------------------------------------------------------------------------------
----------------------
| e22f8b68 |       5000  | e22f8bc4 |       1000    |  e22f8ca0  |        2  |
0  |     0| e2353000  | \Microsoft
\Windows\UsrClass.dat
| e28c3008 |     3fe000  | e1e84000 |       c000    |  e28c3140  |      116  |
0  |     0| e1e48000  | ttings\Adm
inistrator\ntuser.dat
| e23ec008 |       1000  | e23ec064 |          0    |  00000000  |        1  |
0  |     0| e23ee000  | \Microsoft
\Windows\UsrClass.dat
| e23ed760 |      37000  | e23ed7bc |       1000    |  e23ed898  |       14  |
0  |     0| e23ef000  | ettings\Lo
calService\ntuser.dat
...
```

In the preceding output, the Administrator account's profile hive (the full path of which, \Documents and Settings\Administrator\ntuser.dat, is truncated in the output) has 116 mapped views and is approximately 4 MB in size (0x3f000 in decimal). The *!reg viewlist* command will dump the mapped views of the hive you specify. Here's the output of that command when executed for the UsrClass.dat hive that was printed as the first hive of the *!reg hivelist* command's output:

```
kd> !reg viewlist e22f8b68

   0  Pinned Views ; PinViewListHead = e22f8da0 e22f8da0

   2  Mapped Views ; LRUViewListHead = e1cf4448 e1c5d440
-------------------------------------------------------------------------------------
----------------------
| ViewAddr |FileOffset|   Size   |ViewAddress|   Bcb    |    LRUViewList     |    PinV
iewList     | UseCount |
-------------------------------------------------------------------------------------
----------------------
| e1cf4448 |        0 |     4000 | c9a40000  | 8a4bb0e9 | e1c5d440  e22f8d98 | e1cf445
0  e1cf4450 |        0 |
| e1c5d440 |     4000 |     2000 | c9a44000  | 8a4bb0e9 | e22f8d98  e1cf4448 | e1c5d44
8  e1c5d448 |        0 |
-------------------------------------------------------------------------------------
----------------------
```

The output shows the addresses of the two views that the *hivelist* command reported for the hive in the ViewAddress column. Using the debugger's *db* command to dump the contents of memory at the address of the first view reveals that it maps the base block of the hive, recognizable with its *regf* signature:

```
kd> db c9a40000
c9a40000  72 65 67 66 d5 01 00 00-d5 01 00 00 cc 20 43 c7  regf......... C.
c9a40010  3d 40 c4 01 01 00 00 00-03 00 00 00 00 00 00 00  =@..............
c9a40020  01 00 00 00 20 00 00 00-00 50 00 00 01 00 00 00  .... ....P......
c9a40030  5c 00 4d 00 69 00 63 00-72 00 6f 00 73 00 6f 00  \.M.i.c.r.o.s.o.
c9a40040  66 00 74 00 5c 00 57 00-69 00 6e 00 64 00 6f 00  f.t.\.W.i.n.d.o.
c9a40050  77 00 73 00 5c 00 55 00-73 00 72 00 43 00 6c 00  w.s.\.U.s.r.C.l.
c9a40060  61 00 73 00 73 00 2e 00-64 00 61 00 74 00 00 00  a.s.s...d.a.t...
c9a40070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

To deal with noncontiguous memory addresses referencing hive data in memory, the configuration manager adopts a strategy similar to what the Windows memory manager uses to map virtual memory addresses to physical memory addresses. The configuration manager employs a two-level scheme, which Figure 4-7 illustrates, that takes as input a cell index (that is, a hive file offset) and returns as output both the address in memory of the block the cell index resides in and the address in memory of the block the cell resides in. Remember that a bin can contain one or more blocks and that hives grow in bins, so Windows always represents a bin with a contiguous region of memory. Therefore, all blocks within a bin occur within the same cache manager view (in Windows XP and Windows Server 2003) or portion of a paged pool (in Windows 2000).

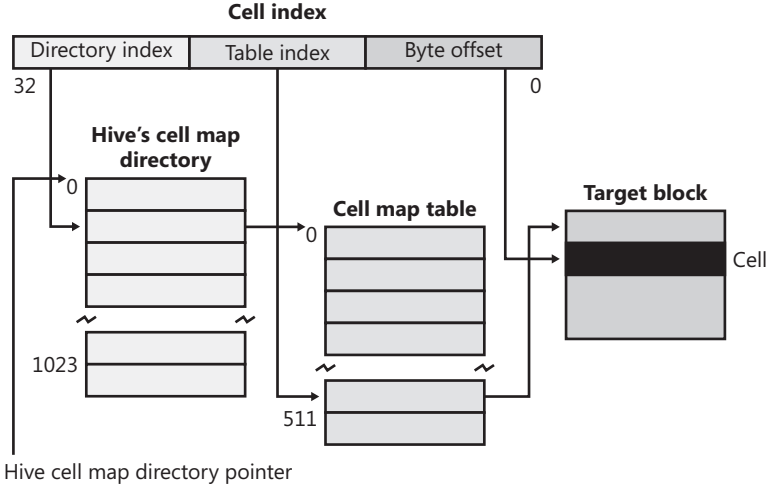


Figure 4-7 Structure of a cell index

To implement the mapping, the configuration manager divides a cell index logically into fields, in the same way that the memory manager divides a virtual address into fields. Windows interprets a cell index's first field as an index into a hive's cell map directory. The cell

map directory contains 1024 entries, each of which refers to a cell map table that contains 512 map entries. An entry in this cell map table is specified by the second field in the cell index. That entry locates the bin and block memory addresses of the cell. In Windows XP and Windows Server 2003, not all bins are necessarily mapped into memory, and if a cell lookup yields an address of 0, the configuration manager maps the bin into memory, unmapping another on the mapping LRU list it maintains, if necessary.

In the final step of the translation process, the configuration manager interprets the last field of the cell index as an offset into the identified block to precisely locate a cell in memory. When a hive initializes, the configuration manager dynamically creates the mapping tables, designating a map entry for each block in the hive, and it adds and deletes tables from the cell directory as the changing size of the hive requires.

The Registry Namespace and Operation

The configuration manager defines a *key object* object type to integrate the registry's namespace with the kernel's general namespace. The configuration manager inserts a key object named *Registry* into the root of the Windows namespace, which serves as the entry point to the registry. Regedit shows key names in the form HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet, but the Windows subsystem translates such names into their object namespace form (for example, \Registry\Machine\System\CurrentControlSet). When the Windows object manager parses this name, it encounters the key object by the name of Registry first and hands the rest of the name to the configuration manager. The configuration manager takes over the name parsing, looking through its internal hive tree to find the desired key or value. Before we describe the flow of control for a typical registry operation, we need to discuss key objects and *key control blocks*. Whenever an application opens or creates a registry key, the object manager gives a handle with which to reference the key to the application. The handle corresponds to a key object that the configuration manager allocates with the help of the object manager. By using the object manager's object support, the configuration manager takes advantage of the security and reference-counting functionality that the object manager provides.

For each open registry key, the configuration manager also allocates a key control block. A key control block stores the full pathname of the key, includes the cell index of the key node that the control block refers to, and contains a flag that notes whether the configuration manager needs to delete the key cell that the key control block refers to when the last handle for the key closes. Windows places all key control blocks into a hash table to enable quick searches for existing key control blocks by name. A key object points to its corresponding key control block, so if two applications open the same registry key, each will receive a key object, and both key objects will point to a common key control block.

When an application opens an existing registry key, the flow of control starts with the application specifying the name of the key in a registry API that invokes the object manager's name-parsing routine. The object manager, upon encountering the configuration manager's registry key object in the namespace, hands the pathname to the configuration manager. The configu-

ration manager uses the in-memory hive data structures to search through keys and subkeys to find the specified key. If the configuration manager finds the key cell, the configuration manager searches the key control block tree to determine whether the key is open (by the same or another application). The search routine is optimized to always start from the closest ancestor with a key control block already opened. For example, if an application opens \Registry\Machine\Key1\Subkey2, and \Registry\Machine is already opened, the parse routine uses the key control block of \Registry\Machine as a starting point. If the key is open, the configuration manager increments the existing key control block's reference count. If the key isn't open, the configuration manager allocates a new key control block and inserts it into the tree. Then the configuration manager allocates a key object, points the key object at the key control block, and returns control to the object manager, which returns a handle to the application.

When an application creates a new registry key, the configuration manager first finds the key cell for the new key's parent. The configuration manager then searches the list of free cells for the hive in which the new key will reside to determine whether cells exist that are large enough to hold the new key cell. If there aren't any free cells large enough, the configuration manager allocates a new bin and uses it for the cell, placing any space at the end of the bin on the free cell list. The new key cell fills with pertinent information—including the key's name—and the configuration manager adds the key cell to the subkey list of the parent key's subkey-list cell. Finally, the system stores the cell index of the parent cell in the new subkey's key cell.

The configuration manager uses a key control block's reference count to determine when to delete the key control block. When all the handles that refer to a key in a key control block close, the reference count becomes 0, which denotes that the key control block is no longer necessary. If an application that calls an API to delete the key sets the delete flag, the configuration manager can delete the associated key from the key's hive because it knows that no application is keeping the key open.

EXPERIMENT: Viewing Key Control Blocks

You can use the kernel debugger to list all the key control blocks allocated on a system with the command *!reg openkeys*. Alternatively, if you want to view the key control block for a particular open key, use *!reg findkcb*:

```
kd> !reg findkcb \registry\machine\software\microsoft

Found KCB = e1034d40 :: \REGISTRY\MACHINE\SOFTWARE\MICROSOFT
```

You can then examine a reported key control block with the *!reg kcb* command:

```
kd> !reg kcb e1034d40

Key             : \REGISTRY\MACHINE\SOFTWARE\MICROSOFT
RefCount        : 1f
Flags           : CompressedName, Stable
```

```
ExtFlags         :
Parent           : 0xe1997368
KeyHive          : 0xe1c8a768
KeyCell          : 0x64e598 [cell index]
TotalLevels      : 4
DelayedCloseIndex: 2048
MaxNameLen       : 0x3c
MaxValueNameLen  : 0x0
MaxValueDataLen  : 0x0
LastWriteTime    : 0x 1c42501:0x7eb6d470
KeyBodyListHead  : 0xe1034d70 0xe1034d70
SubKeyCount      : 137
ValueCache.Count : 0
KCBLock          : 0xe1034d40
KeyLock          : 0xe1034d40
```

The Flags field indicates that the name is stored in compressed form and the SubKeyCount field shows that the key has 137 subkeys.

Stable Storage

To make sure that a nonvolatile registry hive (one with an on-disk file) is always in a recoverable state, the configuration manager uses *log hives*. Each nonvolatile hive has an associated log hive, which is a hidden file with the same base name as the hive and a *.log* extension. For example, if you look in your \Windows\System32\Config directory (and you have the Show Hidden Files And Folders folder option selected), you'll see System.log, Sam.log, and other .log files. When a hive initializes, the configuration manager allocates a bit array in which each bit represents a 512-byte portion, or *sector*, of the hive. This array is called the *dirty sector array* because an *on* bit in the array means that the system has modified the corresponding sector in the hive in memory and must write the sector back to the hive file. (An *off* bit means that the corresponding sector is up to date with the in-memory hive's contents.)

When the creation of a new key or value or the modification of an existing key or value takes place, the configuration manager notes the sectors of the hive that change in the hive's dirty sector array. Then the configuration manager schedules a lazy write operation, or a *hive sync*. The hive lazy writer system thread wakes up 5 seconds after the request to synchronize the hive and writes dirty hive sectors for all hives from memory to the hive files on disk. Thus, the system flushes, at the same time, all the registry modifications that take place between the time a hive sync is requested and the time the hive sync occurs. When a hive sync takes place, the next hive sync will occur no sooner than 5 seconds later.

Note On Windows Server 2003, you can change the default 5-second delay the hive lazy writer thread uses up by setting the registry value HKLM\System\CurrentControlSet\Session Manager\Configuration Manager\RegistryLazyFlushInterval.

If the lazy writer simply wrote all a hive's dirty sectors to the hive file and the system crashed in midoperation, the hive file would be in an inconsistent (corrupted) and unrecoverable state. To prevent such an occurrence, the lazy writer first dumps the hive's dirty sector array and all the dirty sectors to the hive's log file, increasing the log file's size if necessary. The lazy writer then updates a sequence number in the hive's base block and writes the dirty sectors to the hive. When the lazy writer is finished, it updates a second sequence number in the base block. Thus, if the system crashes during the write operations to the hive, at the next reboot the configuration manager will notice that the two sequence numbers in the hive's base block don't match. The configuration manager can update the hive with the dirty sectors in the hive's log file to roll the hive forward. The hive is then up to date and consistent.

To further protect the integrity of the crucial SYSTEM hive in Windows 2000, the configuration manager maintains a mirror of the SYSTEM hive on disk. If you look at the nonhidden files in a Windows 2000 \Windows\System32\Config directory, you'll see System.alt. System.alt is the *alternate hive*. Whenever a hive sync flushes dirty sectors to the SYSTEM hive, the hive sync also updates the System.alt hive. If the configuration manager detects that the SYSTEM hive is corrupt when the system boots, the configuration manager attempts to load the hive's alternate. If that hive is usable, it then uses that alternate to update the original SYSTEM hive.

Windows XP and Windows Server 2003 do not maintain a System.alt hive because NTLDR on those versions of Windows knows how to process the System.log file to bring up to date a System hive that's become inconsistent during a shut down or crash. Windows Server 2003 has other enhancements for tolerating corruption of the registry. Prior to Windows Server 2003, the configuration manager crashes the system if it reads a base block, bin, or cell that contains data that fails basic consistency checks. The configuration manager in Windows Server 2003 is more tolerant of such problems, and if the corruption isn't too severe, it will reinitialize corrupted data structures, possibly deleting subkeys in the process, and continue operation. If it has to resort to self-healing operation, it pops up a system error dialog box notifying the user.

Note When you look at the hidden files on \Windows\System32\Config, you'll also see a file named System.sav. System.Sav is the version of the SYSTEM hive that served as the initial copy of the System hive and is what Windows Setup copied from the install media.

Registry Optimizations

The configuration manager makes a few noteworthy performance optimizations. First, virtually every registry key has a security descriptor that protects access to the key. Storing a unique security-descriptor copy for every key in a hive would be highly inefficient, however, because the same security settings often apply to entire subtrees of the registry. When the system applies security to a key in Windows 2000, the configuration manager first checks the security descriptors associated with the key's parent key and then checks all the parent's subkeys. If any of those security descriptors match the security descriptor the system is applying to the key, the configuration manager simply shares the existing descriptors with the key,

employing a reference count to track how many keys share the same descriptor. In Windows XP and Windows Server 2003, the configuration manager checks a pool of the unique security descriptors used within the same hive as the key to which new security is being applied, and it shares any existing descriptor for the key, ensuring that there is at most one copy of every unique security descriptor in a hive.

The configuration manager also optimizes the way it stores key and value names in a hive. Although the registry is fully Unicode-capable and specifies all names using the Unicode convention, if a name contains only ASCII characters, the configuration manager stores the name in ASCII form in the hive. When the configuration manager reads the name (such as when performing name lookups), it converts the name into Unicode form in memory. Storing the name in ASCII form can significantly reduce the size of a hive.

To minimize memory usage, key control blocks don't store full key registry pathnames. Instead, they reference only a key's name. For example, a key control block that refers to \Registry\System\Control would refer to the name *Control* rather than to the full path. A further memory optimization is that the configuration manager uses key name control blocks to store key names, and all key control blocks for keys with the same name share the same key name control block. To optimize performance, the configuration manager stores the key control block names in a hash table for quick lookups.

To provide fast access to key control blocks, the configuration manager stores frequently accessed key control blocks in the cache table, which is configured as a hash table. When the configuration manager needs to look up a key control block, it first checks the cache table. Finally, the configuration manager has another cache, the delayed close table, that stores key control blocks that applications close, so that an application can quickly reopen a key it has recently closed. The configuration manager removes the oldest key control blocks from the delayed close table as it adds the most recently closed blocks to the table.

Services

Almost every operating system has a mechanism to start processes at system startup time that provide services not tied to an interactive user. In Windows, such processes are called *services* or *Windows services*, because they rely on the Windows API to interact with the system. Services are similar to UNIX daemon processes and often implement the server side of client/server applications. An example of a Windows service might be a Web server because it must be running regardless of whether anyone is logged on to the computer and it must start running when the system starts so that an administrator doesn't have to remember, or even be present, to start it.

Windows services consist of three components: a service application, a service control program (SCP), and the service control manager (SCM). First, we'll describe service applications, service accounts, and the operations of the SCM. Then we'll explain how auto-start services are started during the system boot. We'll also cover the steps the SCM takes when a service fails during its startup and the way the SCM shuts down services.

Service Applications

Service applications, such as Web servers, consist of at least one executable that runs as a Windows service. A user wanting to start, stop, or configure a service uses an SCP. Although Windows supplies built-in SCPs that provide general start, stop, pause, and continue functionality, some service applications include their own SCP that allows administrators to specify configuration settings particular to the service they manage.

Service applications are simply Windows executables (GUI or console) with additional code to receive commands from the SCM as well as to communicate the application's status back to the SCM. Because most services don't have a user interface, they are built as console programs.

When you install an application that includes a service, the application's setup program must register the service with the system. To register the service, the setup program calls the Windows *CreateService* function, a services-related function implemented in Advapi32.dll (\Windows\System32\Advapi32.dll). Advapi32, the "Advanced API" DLL, implements all the client-side SCM APIs.

When a setup program registers a service by calling *CreateService*, a message is sent to the SCM on the machine where the service will reside. The SCM then creates a registry key for the service under HKLM\SYSTEM\CurrentControlSet\Services. The Services key is the nonvolatile representation of the SCM's database. The individual keys for each service define the path of the executable image that contains the service as well as parameters and configuration options.

After creating a service, an installation or management application can start the service via the *StartService* function. Because some service-based applications also must initialize during the boot process to function, it's not unusual for a setup program to register a service as an autostart service, ask the user to reboot the system to complete an installation, and let the SCM start the service as the system boots.

When a program calls *CreateService*, it must specify a number of parameters describing the service's characteristics. The characteristics include the service's type (whether it's a service that runs in its own process rather than a service that shares a process with other services), the location of the service's executable image file, an optional display name, an optional account name and password used to start the service in a particular account's security context, a start type that indicates whether the service starts automatically when the system boots or manually under the direction of an SCP, an error code that indicates how the system should react if the service detects an error when starting, and, if the service starts automatically, optional information that specifies when the service starts relative to other services.

The SCM stores each characteristic as a value in the service's registry key. Figure 4-8 shows an example of a service registry key.

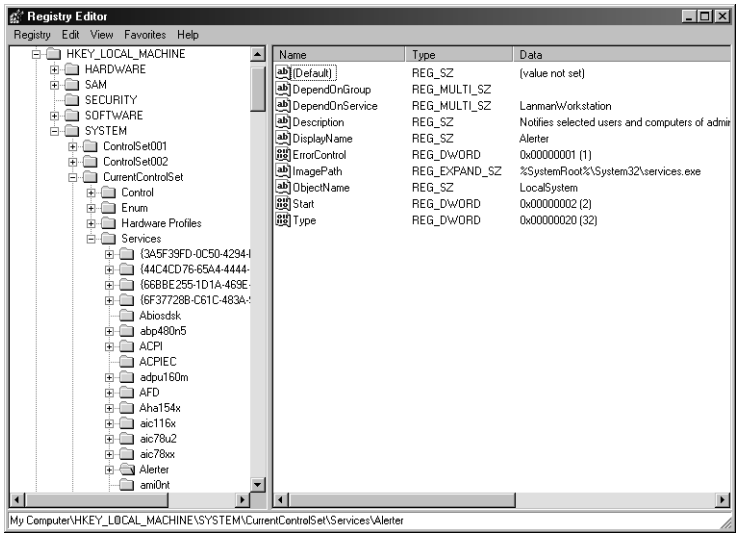


Figure 4-8 Example of a service registry key

Table 4-7 lists all the service characteristics, many of which also apply to device drivers. (Not every characteristic applies to every type of service or device driver.) If a service needs to store configuration information that is private to the service, the convention is to create a subkey named Parameters under its service key and then store the configuration information in values under that subkey. The service then can retrieve the values by using standard registry functions.

Note The SCM does not access a service's Parameters subkey until the service is deleted, at which time the SCM deletes the service's entire key, including subkeys like Parameters.

Table 4-7 Service and Driver Registry Parameters

Value Setting	Value Name	Value Setting Description
Start	SERVICE_BOOT_START (0)	Ntldr or Osloader preloads the driver so that it is in memory during the boot. These drivers are initialized just prior to SERVICE_ SYSTEM_START drivers.
	SERVICE_SYSTEM_START (1)	The driver loads and initializes during kernel initialization after SERVICE_ BOOT_START drivers have initialized.
	SERVICE_AUTO_START (2)	The SCM starts the driver or service after the SCM process, Services.exe, starts.
	SERVICE_DEMAND_START (3)	The SCM starts the driver or service on demand.

Table 4-7 Service and Driver Registry Parameters

Value Setting	Value Name	Value Setting Description
	SERVICE_DISABLED (4)	The driver or service doesn't load or initialize.
ErrorControl	SERVICE_ERROR_IGNORE (0)	Any error the driver or service returns is ignored and no warning is logged or displayed.
	SERVICE_ERROR_NORMAL (1)	If the driver or service reports an error, a warning displays.
	SERVICE_ERROR_SEVERE (2)	If the driver or service returns an error and last known good isn't being used, reboot into last known good; otherwise, continue the boot.
	SERVICE_ERROR_CRITICAL (3)	If the driver or service returns an error and last known good isn't being used, reboot into last known good; otherwise, stop the boot with a blue screen crash.
Type	SERVICE_KERNEL_DRIVER (1)	Device driver.
	SERVICE_FILE_SYSTEM_DRIVER (2)	Kernel-mode file system driver.
	SERVICE_ADAPTER (4)	Obsolete.
	SERVICE_RECOGNIZER_DRIVER (8)	File system recognizer driver.
	SERVICE_WIN32_OWN_PROCESS (16)	The service runs in a process that hosts only one service.
	SERVICE_WIN32_SHARE_PROCESS (32)	The service runs in a process that hosts multiple services.
	SERVICE_INTERACTIVE_PROCESS (256)	The service is allowed to display windows on the console and receive user input.
Group	Group name	The driver or service initializes when its group is initialized.
Tag	Tag number	The specified location in a group initialization order. This parameter doesn't apply to services.
ImagePath	Path to service or driver executable file	If *ImagePath* isn't specified, the I/O manager looks for drivers in \Windows\System32\Drivers and the SCM uses Windows functions that search for the image using the PATH environment variable.
DependOnGroup	Group name	The driver or service won't load unless a driver or service from the specified group loads.

Table 4-7 Service and Driver Registry Parameters

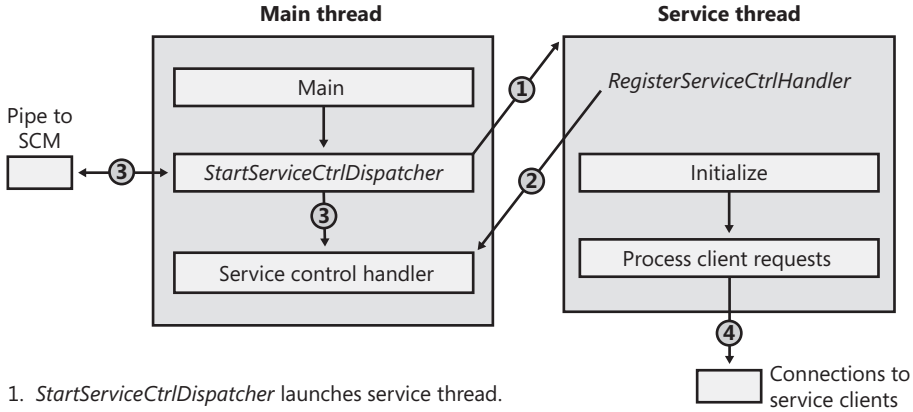
Value Setting	Value Name	Value Setting Description
DependOnService	Service name	The service won't load until after the specified service loads. This parameter doesn't apply to device drivers other than those with a start type of SERVICE_AUTO_START.
ObjectName	Usually LocalSystem, but can be an account name, such as .\Administrator	Specifies the account in which the service will run. If ObjectName isn't specified, LocalSystem is the account used. This parameter doesn't apply to device drivers.
DisplayName	Name of service	The service application shows services by this name. If no name is specified, the name of the service's registry key becomes its name.
Description	Description of service	Up to 32767-byte description of the service.
FailureActions	Description of actions the SCM should take when service process exits unexpectedly	Failure actions include restarting the service process, rebooting the system, and running a specified program. This value doesn't apply to drivers.
FailureCommand	Program command line	The SCM reads this value only if FailureActions specifies that a program should execute upon service failure. This value doesn't apply to drivers.
Security	Security descriptor	This value contains the security descriptor that defines who has what access to the service object created internally by the SCM.

Notice that Type values include three that apply to device drivers: device driver, file system driver, and file system recognizer. These are used by Windows device drivers, which also store their parameters as registry data in the Services registry key. The SCM is responsible for starting drivers with a Start value of SERVICE_AUTO_START or SERVICE_DEMAND_START, so it's natural for the SCM database to include drivers. Services use the other types, SERVICE_WIN32_OWN_PROCESS and SERVICE_WIN32_SHARE_PROCESS, which are mutually exclusive. An executable that hosts more than one service specifies the SERVICE_WIN32_SHARE_PROCESS type. An advantage to having a process run more than one service is that the system resources that would otherwise be required to run them in distinct processes are saved. A potential disadvantage is that if one of the services of a collection running in the same process causes an error that terminates the process, all the services of that process terminate. Also, another limitation is that all the services must run under the same account.

When the SCM starts a service process, the process immediately invokes the *StartServiceCtrlDispatcher* function. *StartServiceCtrlDispatcher* accepts a list of entry points into services, one entry point for each service in the process. Each entry point is identified by the name of the service the entry point corresponds to. After making a named pipe communications connection to the SCM, *StartServiceCtrlDispatcher* sits in a loop waiting for commands to come through the pipe from the SCM. The SCM sends a service-start command each time it starts a service the process owns. For each start command it receives, the *StartServiceCtrlDispatcher* function creates a thread, called a service thread, to invoke the starting service's entry point and implement the command loop for the service. *StartServiceCtrlDispatcher* waits indefinitely for commands from the SCM and returns control to the process's main function only when all the process's services have stopped, allowing the service process to clean up resources before exiting.

A service entry point's first action is to call the *RegisterServiceCtrlHandler* function. This function receives and stores a pointer to a function, called the *control handler*, which the service implements to handle various commands it receives from the SCM. *RegisterServiceCtrlHandler* doesn't communicate with the SCM, but it stores the function in local process memory for the *StartServiceCtrlDispatcher* function. The service entry point continues initializing the service, which can include allocating memory, creating communications end points, and reading private configuration data from the registry. A convention most services follow is to store their parameters under a subkey of their service registry key, named Parameters. While the entry point is initializing the service, it might periodically send status messages, using the *SetServiceStatus* function, to the SCM indicating how the service's startup is progressing. After the entry point finishes initialization, a service thread usually sits in a loop waiting for requests from client applications. For example, a Web server would initialize a TCP listen socket and wait for inbound HTTP connection requests.

A service process's main thread, which executes in the *StartServiceCtrlDispatcher* function, receives SCM commands directed at services in the process and invokes the target service's control handler function (stored by *RegisterServiceCtrlHandler*). SCM commands include stop, pause, resume, interrogate, and shutdown, or application-defined commands. Figure 4-9 shows the internal organization of a service process. Pictured are the two threads that make up a process hosting one service: the main thread and the service thread.



1. *StartServiceCtrlDispatcher* launches service thread.
2. Service thread registers control handler.
3. *StartServiceCtrlDispatcher* calls handlers in response to SCM commands.
4. Service thread processes client requests.

Figure 4-9 Inside a service process

SrvAny Tool

If you have a program that you want to run as a service, you need to modify the startup code to conform to the requirements for services outlined in this section. If you don't have the source code, you can use the SrvAny tool in the Windows resource kits. SrvAny enables you to run any application as a service. It reads the path of the service file that it must load from the Parameters subkey of the service's registry key. When SrvAny starts, it notifies the SCM that it is hosting a particular service, and when it receives a start command, it launches the service executable as a child process. The child process receives a copy of the SrvAny process's access token and a reference to the same window station, so the executable runs within the same security account and with the same interactivity setting as you specified when configuring the SrvAny process. SrvAny services don't have the share-process Type value, so each application you install as a service with SrvAny runs in a separate process with a different instance of the SrvAny host program.

Service Accounts

The security context of a service is an important consideration for service developers as well as for system administrators because it dictates what resources the process can access. Unless a service installation program or administrator specifies otherwise, most services run in the security context of the *local system account* (displayed sometimes as SYSTEM and other times as LocalSystem). Windows XP introduced two variants on the local system account, the *network service* and *local service* accounts. The new accounts have fewer capabilities than the local system account from a security standpoint, and any built-in Windows service that does not

require the power of the local system account runs in the appropriate alternate service account. The following subsections describe the special characteristics of these accounts.

The Local System Account

The local system account is the same account in which core Windows user-mode operating system components run, including the Session Manager (\Windows\System32\Smss.exe), the Windows subsystem process (Csrss.exe), the local security authority subsystem (\Windows\System32\Lsass.exe), and the Winlogon process (\Windows\System32\Winlogon.exe).

From a security perspective, the local system account is extremely powerful—more powerful than any local or domain account when it comes to security ability on a local system. This account has the following characteristics:

- It is a member of the local administrators group. Table 4-8 shows the groups to which the local system account belongs. (See Chapter 8 for information on how group membership is used in object access checks.)
- It has the right to enable virtually every privilege (even privileges not normally granted to the local administrator account, such as creating security tokens). See Table 4-9 for the list of privileges assigned to the local system account. (Chapter 8 describes the use of each privilege.)
- Most files and registry keys grant full access to the local system account. (Even if they don't grant full access, a process running under the local system account can exercise the take-ownership privilege to gain access.)
- Processes running under the local system account run with the default user profile (HKU\.DEFAULT). Therefore, they can't access configuration information stored in the user profiles of other accounts.
- When a system is a member of a Windows domain, the local system account includes the machine security identifier (SID) for the computer on which a service process is running. Therefore, a service running in the local system account will be automatically authenticated on other machines in the same forest by using its computer account. (A *forest* is a grouping of domains.)
- Unless the machine account is specifically granted access to resources (such as network shares, named pipes, and so on), a process can access network resources that allow null sessions—that is, connections that require no credentials. You can specify the shares and pipes on a particular computer that permit null sessions in the NullSessionPipes and NullSessionShares registry values under HKLM\SYSTEM\CurrentControlSet\Services\lanmanserver\parameters.

Table 4-8 Service Account Group Membership

Local System	Network Service	Local Service
Everyone	Everyone	Everyone
Authenticated Users	Authenticated Users	Authenticated Users
Administrators	Users	Users
	Local	Local
	Network Service	Local Service
	Service	Service

Table 4-9 Service Account Privileges

Local System	Network Service	Local Service
SeAssignPrimaryToken Privilege	SeAssignPrimaryToken Privilege	SeAssignPrimaryTokenPrivilege
SeAuditPrivilege	SeAuditPrivilege	SeAuditPrivilege
SeBackupPrivilege	SeChangeNotifyPrivilege	SeChangeNotifyPrivilege
SeChangeNotifyPrivilege	SeIncreaseQuotaPrivilege	SeIncreaseQuotaPrivilege
SeCreateGlobalPrivilege	Privileges assigned to the Everyone, Authenticated Users, and Users groups	Privileges assigned to the Everyone, Authenticated Users, and Users groups
SeCreatePagefilePrivilege		
SeCreatePermanentPrivilege		
SeCreateTokenPrivilege*		
SeDebugPrivilege		
SeImpersonatePrivilege		
SeIncreaseBasePriorityPrivilege		
SeIncreaseQuotaPrivilege		
SeLoadDriverPrivilege		
SeLockMemoryPrivilege		
SeManageVolumePrivilege		
SeProfileSingleProcessPrivilege		
SeRestorePrivilege		
SeSecurityPrivilege		
SeShutdownPrivilege		
SeSystemEnvironmentPrivilege		
SeSystemTimePrivilege		
SeTakeOwnershipPrivilege		
SeTcbPrivilege		
SeUndockPrivilege		

* The local system account on Windows Server 2003 does not include this privilege.

The Network Service Account

The network service account is intended for use by services that wish to authenticate to other machines on the network using the computer account, as does the local system account, but do not have the need for membership in the administrators group or the use of many of the privileges assigned to the local system account. Because the network service account does not belong to the administrators group, services running in the network service account by default have access to far fewer registry keys and file system folders and files than the services running the local system account. Further, the assignment of few privileges limits the scope of a compromised network service process. For example, a process running in the network service account cannot load a device driver or open arbitrary processes.

Another difference between the network service and local system accounts is that processes running in the network service account use the network service account's profile. The registry component of the network service profile loads under HKU\S-1-5-20, and the files and directories that make up the component reside in \Documents and Settings\NetworkService.

A service that runs in the network service account in Windows XP and Windows Server 2003 is the DNS client, which is responsible for resolving DNS names and for locating domain controllers.

The Local Service Account

The local service account is virtually identical to the network service account with the important difference that it can access only network resources that allow anonymous access. Table 4-9 shows that it has the same privileges as the local service account, and Table 4-8 shows that it belongs to the same groups with the exception that it belongs to the Network Service group instead of the Local Service group. The profile used by processes running in the local service loads into HKU\S-1-5-19 and is stored in \Documents and Settings\LocalService.

Examples of services that Windows XP and Windows Server 2003 run in the local service account include the Remote Registry Service that allows remote access to the local system's registry, the Alerter service that receives network-broadcast administrative alerts messages, and the LmHosts service that performs NetBIOS name resolution.

Running Services in Alternate Accounts

Because of the restrictions just outlined, some services need to run with the security credentials of a user account. You can configure a service to run in an alternate account when the service is created or by specifying an account and password that the service should run under with the Windows Services MMC snap-in. In the Services snap-in, right-click on a service and select Properties, click the Log On tab, and select the This Account option, as shown in Figure 4-10.

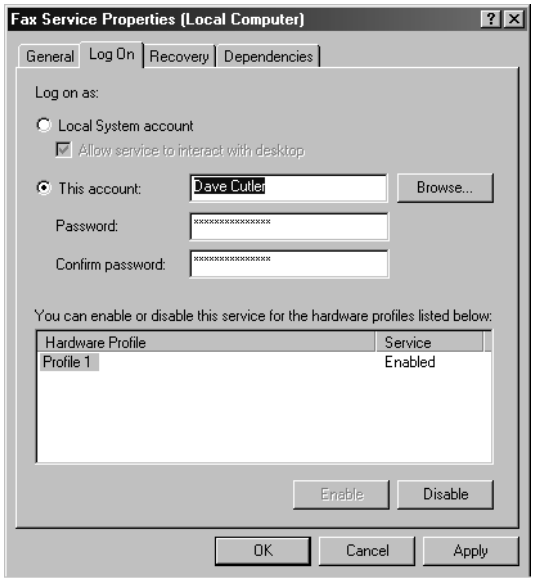


Figure 4-10 Service account settings

Interactive Services

Another restriction for services running under the local system, local service, and network service accounts is that they can't (without using a special flag on the *MessageBox* function, discussed in a moment) display dialog boxes or windows on the interactive user's desktop. This limitation isn't the direct result of running under these accounts but rather a consequence of the way the Windows subsystem assigns service processes to window stations.

The Windows subsystem associates every Windows process with a *window station*. A window station contains desktops, and desktops contain windows. Only one window station can be visible on a console and receive user mouse and keyboard input. In a Terminal Services environment, one window station per session is visible, but services all run as part of the console session. Windows names the visible window station WinSta0, and all interactive processes access WinSta0.

Unless otherwise directed, the Windows subsystem associates services running in the local system account with a nonvisible window station named Service-0x0-3e7$ that all noninteractive services share. The number in the name, *3e7*, represents the logon session identifier Lsass assigns to the logon session the SCM uses for noninteractive services running in the local system account.

Services configured to run under a user account (that is, not the local system account) are run in a different nonvisible window station named with the LSASS logon identifier assigned for the service's logon session. Figure 4-11 shows a sample display from the Winobj tool, available from *www.sysinternals.com*, viewing the object manager directory in which Windows

places window station objects. Visible are the interactive window station (WinSta0), the noninteractive system service window station (Service-0x0-3e7$), and a noninteractive window station assigned to a service process logged on as a user (Service-0x0-6368f$).

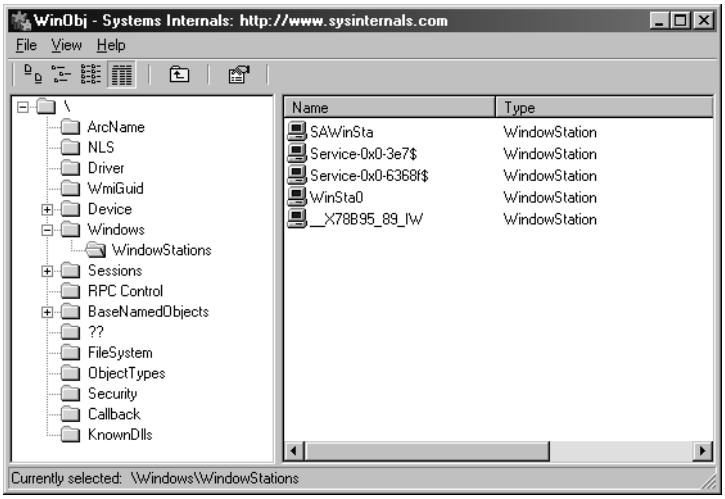


Figure 4-11 List of window stations

Regardless of whether services are running in a user account, the local system account, or the local or network service accounts, services that aren't running on the visible window station can't receive input from a user or display windows on the console. In fact, if a service were to pop up a normal dialog box on the window station, the service would appear hung because no user would be able to see the dialog box, which of course would prevent the user from providing keyboard or mouse input to dismiss it and allow the service to continue executing. (The one exception is if the special flag MB_SERVICE_NOTIFICATION or MB_DEFAULT_DESKTOP_ ONLY is set on the *MessageBox* call—if MB_SERVICE_NOTIFICATION is specified, the message box will always be displayed on the interactive window station, even if the service wasn't configured with permission to interact with the user; if MB_DEFAULT_DESKTOP_ONLY is specified, the message box is displayed on the default desktop of the interactive window station.)

In rare cases, a service can have a valid reason to interact with the user via dialog boxes or windows. To configure a service with the right to interact with the user, the SERVICE_ INTERACTIVE_PROCESS modifier must be present in the service's registry key's Type parameter. (Note that services configured to run under a user account can't be marked as interactive.) When the SCM starts a service marked as interactive, it launches the service's process in the local system account's security context but connects the service with WinSta0 instead of the noninteractive service window station. This connection to WinSta0 allows the service to display dialog boxes and windows on the console and allows those windows to respond to user input.

> **Note** Microsoft discourages running interactive services, especially in the local system account, because of the inherent security vulnerability it creates. Windows presented by an interactive service are susceptible to the receipt of windows messages that a malicious process running on the desktop of an unprivileged user can use to cause buffer overflows in the service process and subvert the service process to elevate the security privileges of the malicious process.

The Service Control Manager

The SCM's executable file is \Windows\System32\Services.exe, and like most service processes, it runs as a Windows console program. The Winlogon process starts the SCM early during the system boot. (Refer to Chapter 5 for details on the boot process.) The SCM's startup function, *SvcCtrlMain*, orchestrates the launching of services that are configured for automatic startup. *SvcCtrlMain* executes shortly after the screen switches to a blank desktop but generally before Winlogon has loaded the graphical identification and authentication interface (GINA) that presents a logon dialog box.

SvcCtrlMain first creates a synchronization event named SvcCtrlEvent_ A3752DX that it initializes as nonsignaled. Only after the SCM completes steps necessary to prepare it to receive commands from SCPs does the SCM set the event to a signaled state. The function that an SCP uses to establish a dialog with the SCM is *OpenSCManager*. *OpenSCManager* prevents an SCP from trying to contact the SCM before the SCM has initialized by waiting for SvcCtrlEvent_A3752DX to become signaled.

Next, *SvcCtrlMain* gets down to business and calls *ScCreateServiceDB*, the function that builds the SCM's internal service database. *ScCreateServiceDB* reads and stores the contents of HKLM\SYSTEM\CurrentControlSet\Control\ServiceGroupOrder\List, a REG_MULTI_SZ value that lists the names and order of the defined service groups. A service's registry key contains an optional Group value if that service or device driver needs to control its startup ordering with respect to services from other groups. For example, the Windows networking stack is built from the bottom up, so networking services must specify Group values that place them later in the startup sequence than networking device drivers. SCM internally creates a group list that preserves the ordering of the groups it reads from the registry. Groups include (but are not limited to) NDIS, TDI, Primary Disk, Keyboard Port, and Keyboard Class. Add-on and third-party applications can even define their own groups and add them to the list. Microsoft Transaction Server, for example, adds a group named MS Transactions.

ScCreateServiceDB then scans the contents of HKLM\SYSTEM\CurrentControlSet\Services, creating an entry in the service database for each key it encounters. A database entry includes all the service-related parameters defined for a service as well as fields that track the service's status. The SCM adds entries for device drivers as well as for services because the SCM starts services and drivers marked as auto-start and detects startup failures for drivers marked boot-start and system-start. It also provides a means for applications to query the status of drivers.

The I/O manager loads drivers marked boot-start and system-start before any user-mode processes execute, and therefore any drivers having these start types load before the SCM starts.

ScCreateServiceDB reads a service's Group value to determine its membership in a group and associates this value with the group's entry in the group list created earlier. The function also reads and records in the database the service's group and service dependencies by querying its DependOnGroup and DependOnService registry values. Figure 4-12 shows how the SCM organizes the service entry and group order lists. Notice that the service list is alphabetically sorted. The reason this list is sorted alphabetically is that the SCM creates the list from the Services registry key, and Windows stores registry keys alphabetically.

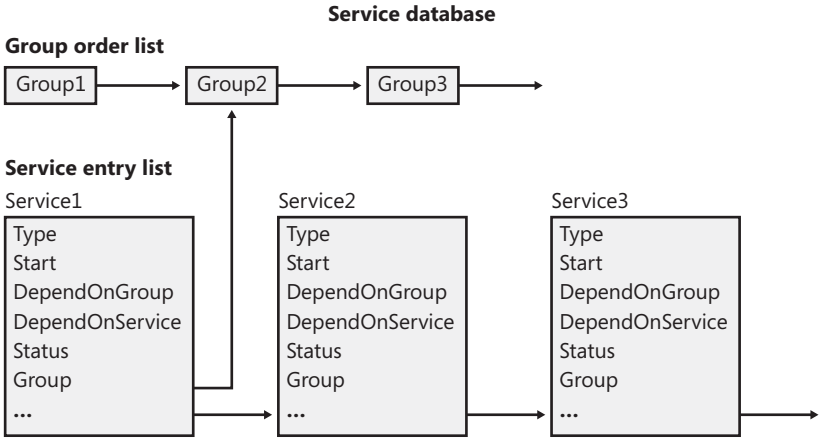


Figure 4-12 Organization of a service database

During service startup, the SCM might need to call on LSASS (for example, to log on a service in a user account), so the SCM waits for LSASS to signal the LSA_RPC_SERVER_ACTIVE synchronization event, which it does when it finishes initializing. Winlogon also starts the LSASS process, so the initialization of LSASS is concurrent with that of the SCM, and the order in which LSASS and the SCM complete initialization can vary. Then *SvcCtrlMain* calls *ScGetBootAndSystemDriverState* to scan the service database looking for boot-start and system-start device driver entries. *ScGetBootAndSystemDriverState* determines whether or not a driver successfully started by looking up its name in the object manager namespace directory named \Driver. When a device driver successfully loads, the I/O manager inserts the driver's object in the namespace under this directory, so if its name isn't present, it hasn't loaded. Figure 4-13 shows Winobj displaying the contents of the Driver directory. If a driver isn't loaded, the SCM looks for its name in the list of drivers returned by the *PnP_DeviceList* function. *PnP_DeviceList* supplies the drivers included in the system's current hardware profile. *SvcCtrlMain* notes the names of drivers that haven't started and that are part of the current profile in a list named ScFailedDrivers.

Before starting the auto-start services, the SCM performs a few more steps. It creates its remote procedure call (RPC) named pipe, which is named \Pipe\Ntsvcs, and then RPC launches a

thread to listen on the pipe for incoming messages from SCPs. The SCM then signals its initialization-complete event, SvcCtrlEvent_A3752DX. Registering a console application shutdown event handler and registering with the Windows subsystem process via *RegisterServiceProcess* prepares the SCM for system shutdown.

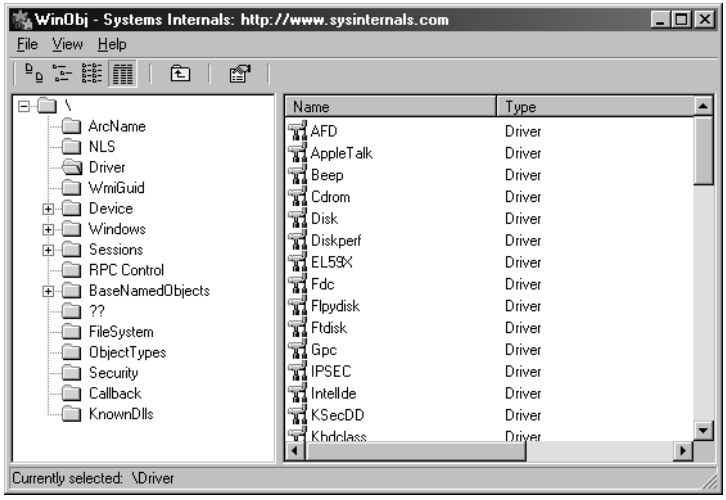


Figure 4-13 List of driver objects

Network Drive Letters

In addition to its role as an interface to services, the SCM has another totally unrelated responsibility: it notifies GUI applications in a system whenever the system creates or deletes a network drive-letter connection. The SCM waits for the Multiple Provider Router (MPR) to signal a named event, \BaseNamedObjects\ScNetDrvMsg, which MPR signals whenever an application assigns a drive letter to a remote network share or deletes a remote-share drive-letter assignment. (See Chapter 13 for more information on MPR.) When MPR signals the event, the SCM calls the *GetDriveType* Windows function to query the list of connected network drive letters. If the list changes across the event signal, the SCM sends a Windows broadcast message of type WM_DEVICECHANGE. The SCM uses either DBT_ DEVICEREMOVECOMPLETE or DBT_DEVICEARRIVAL as the message's subtype. This message is primarily intended for Windows Explorer so that it can update any open My Computer windows to show the presence or absence of a network drive letter.

Service Startup

SvcCtrlMain invokes the SCM function *ScAutoStartServices* to start all services that have a Start value designating auto-start. *ScAutoStartServices* also starts auto-start device drivers. To avoid confusion, you should assume that the term *services* means *services and drivers* unless indi-

cated otherwise. The algorithm in *ScAutoStartServices* for starting services in the correct order proceeds in phases, whereby a phase corresponds to a group and phases proceed in the sequence defined by the group ordering stored in the HKLM\SYSTEM\CurrentControlSet\Control\ServiceGroupOrder\List registry value. The List value, shown in Figure 4-14, includes the names of groups in the order that the SCM should start them. Thus, assigning a service to a group has no effect other than to fine-tune its startup with respect to other services belonging to different groups.

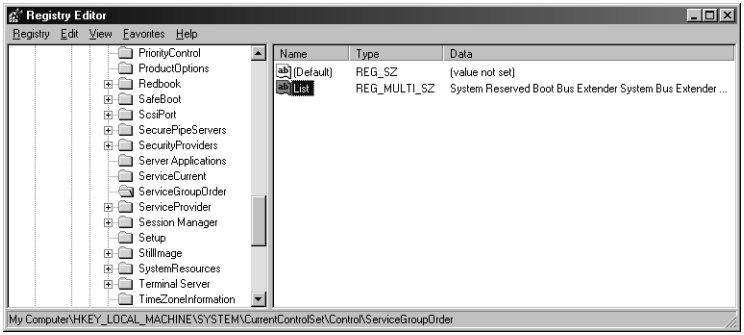


Figure 4-14 ServiceGroupOrder registry key

When a phase starts, *ScAutoStartServices* marks all the service entries belonging to the phase's group for startup. Then *ScAutoStartServices* loops through the marked services seeing whether it can start each one. Part of the check it makes consists of determining whether the service has a dependency on another group, as specified by the existence of the DependOnGroup value in the service's registry key. If a dependency exists, the group on which the service is dependent must have already initialized, and at least one service of that group must have successfully started. If the service depends on a group that starts later than the service's group in the group startup sequence, the SCM notes a "circular dependency" error for the service. If *ScAutoStartServices* is considering a Windows service or an auto-start device driver, it next checks to see whether the service depends on one or more other services, and if so, if those services have already started. Service dependencies are indicated with the DependOnService registry value in a service's registry key. If a service depends on other services that belong to groups that come later in the ServiceGroupOrder\List, the SCM also generates a "circular dependency" error and doesn't start the service. If the service depends on any services from the same group that haven't yet started, the service is skipped.

When the dependencies of a service have been satisfied, *ScAutoStartServices* makes a final check to see whether the service is part of the current boot configuration before starting the service. When the system is booted in safe mode, the SCM ensures that the service is either identified by name or by group in the appropriate safe boot registry key. There are two safe boot keys, Minimal and Network, under HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot, and the one that the SCM checks depends on what safe mode the user booted. If the user chose Safe Mode or Safe Mode With Command Prompt at the special boot menu (which you can access by pressing F8 when prompted in the boot process), the SCM references the

Minimal key; if the user chose Safe Mode With Networking, the SCM refers to Network. The existence of a string value named Option under the SafeBoot key indicates not only that the system booted in safe mode but also the type of safe mode the user selected. For more information about safe boots, see the section "Safe Mode" in Chapter 5.

Once the SCM decides to start a service, it calls *ScStartService*, which takes different steps for services than for device drivers. When *ScStartService* starts a Windows service, it first determines the name of the file that runs the service's process by reading the ImagePath value from the service's registry key. It then examines the service's Type value, and if that value is SERVICE_WINDOWS_SHARE_PROCESS (0x20), the SCM ensures that the process the service runs in, if already started, is logged on using the same account as specified for the service being started. A service's ObjectName registry value stores the user account in which the service should run. A service with no ObjectName or an ObjectName of LocalSystem runs in the local system account.

The SCM verifies that the service's process hasn't already been started in a different account by checking to see whether the service's ImagePath value has an entry in an internal SCM database called the *image database*. If the image database doesn't have an entry for the ImagePath value, the SCM creates one. When the SCM creates a new entry, it stores the logon account name used for the service and the data from the service's ImagePath value. The SCM requires services to have an ImagePath value. If a service doesn't have an ImagePath value, the SCM reports an error stating that it couldn't find the service's path and isn't able to start the service. If the SCM locates an existing image database entry with matching ImagePath data, the SCM ensures that the user account information for the service it's starting is the same as the information stored in the database entry—a process can be logged on as only one account, so the SCM reports an error when a service specifies a different account name than another service that has already started in the same process.

The SCM calls *ScLogonAndStartImage* to log on a service if the service's configuration specifies and to start the service's process. The SCM logs on services that don't run in the system account by calling the LSASS function *LsaLogonUser*. *LsaLogonUser* normally requires a password, but the SCM indicates to LSASS that the password is stored as a service's LSASS "secret" under the key HKLM\SECURITY\Policy\Secrets in the registry. (Keep in mind that the contents of the SECURITY aren't typically visible because its default security settings permit access only from the system account.) When the SCM calls *LsaLogonUser*, it specifies a service logon as the logon type, so LSASS looks up the password in the Secrets subkey that has a name in the form _SC_<service name>.

The SCM directs LSASS to store a logon password as a secret using the *LsaStorePrivateData* function when an SCP configures a service's logon information. When a logon is successful, *LsaLogonUser* returns a handle to an access token to the caller. Windows uses access tokens to represent a user's security context, and the SCM later associates the access token with the process that implements the service.

After a successful logon, the SCM loads the account's profile information, if it's not already loaded, by calling the UserEnv DLL's (\Windows\System32\Userenv.dll) *LoadUserProfile* function. The value HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Profile-List\<user profile key>\ProfileImagePath contains the location on disk of a registry hive that *LoadUserProfile* loads into the registry, making the information in the hive the HKEY_CURRENT_USER key for the service.

An interactive service must open the WinSta0 window station, but before *ScLogonAndStartImage* allows an interactive service to access WinSta0 it checks to see whether the value HKLM\SYSTEM\CurrentControlSet\Control\Windows\NoInteractiveServices is set. Administrators set this value to prevent services marked as interactive from displaying windows on the console. This option is desirable in unattended server environments in which no user is present to respond to popups from interactive services.

As its next step, *ScLogonAndStartImage* proceeds to launch the service's process, if the process hasn't already been started (for another service, for example). The SCM starts the process in a suspended state with the *CreateProcessAsUser* Windows function. The SCM next creates a named pipe through which it communicates with the service process, and it assigns the pipe the name \Pipe\Net\NtControlPipeX, where *X* is a number that increments each time the SCM creates a pipe. The SCM resumes the service process via the *ResumeThread* function and waits for the service to connect to its SCM pipe. If it exists, the registry value HKLM\SYSTEM\CurrentControlSet\Control\ServicesPipeTimeout determines the length of time that the SCM waits for a service to call *StartServiceCtrlDispatcher* and connect before it gives up, terminates the process, and concludes that the service failed to start. If ServicesPipeTimeout doesn't exist, the SCM uses a default timeout of 30 seconds. The SCM uses the same timeout value for all its service communications.

When a service connects to the SCM through the pipe, the SCM sends the service a start command. If the service fails to respond positively to the start command within the timeout period, the SCM gives up and moves on to start the next service. When a service doesn't respond to a start request, the SCM doesn't terminate the process, as it does when a service doesn't call *StartServiceCtrlDispatcher* within the timeout; instead, it notes an error in the system Event Log that indicates the service failed to start in a timely manner.

If the service the SCM starts with a call to *ScStartService* has a Type registry value of SERVICE_KERNEL_DRIVER or SERVICE_FILE_SYSTEM_ DRIVER, the service is really a device driver, and so *ScStartService* calls *ScLoadDeviceDriver* to load the driver. *ScLoadDeviceDriver* enables the load driver security privilege for the SCM process and then invokes the kernel service *NtLoadDriver*, passing in the data in the ImagePath value of the driver's registry key. Unlike services, drivers don't need to specify an ImagePath value, and if the value is absent, the SCM builds an image path by appending the driver's name to the string \Windows\System32\Drivers\.

ScAutoStartServices continues looping through the services belonging to a group until all the services have either started or generated dependency errors. This looping is the SCM's way of automatically ordering services within a group according to their DependOnService dependencies. The SCM will start the services that other services depend on in earlier loops, skipping the dependent services until subsequent loops. Note that the SCM ignores Tag values for Windows services, which you might come across in subkeys under the HKLM\SYSTEM\CurrentControlSet\Services key; the I/O manager honors Tag values to order device driver startup within a group for boot and system-start drivers.

Once the SCM completes phases for all the groups listed in the ServiceGroupOrder\List value, it performs a phase for services belonging to groups not listed in the value and a final phase for services without a group. When it's finished starting all auto-start services and drivers, the SCM signals the event \BaseNamedObjects\SC_AutoStartComplete.

Startup Errors

If a driver or a service reports an error in response to the SCM's startup command, the ErrorControl value of the service's registry key determines how the SCM reacts. If the ErrorControl value is SERVICE_ERROR_IGNORE (0) or the ErrorControl value isn't specified, the SCM simply ignores the error and continues processing service startups. If the ErrorControl value is SERVICE_ERROR_NORMAL (1), the SCM writes an event to the system Event Log that says, "The <service name> service failed to start due to the following error:". The SCM includes the textual representation of the Windows error code that the service returned to the SCM as the reason for the startup failure in the Event Log record. Figure 4-15 shows the Event Log entry that reports a service startup error.

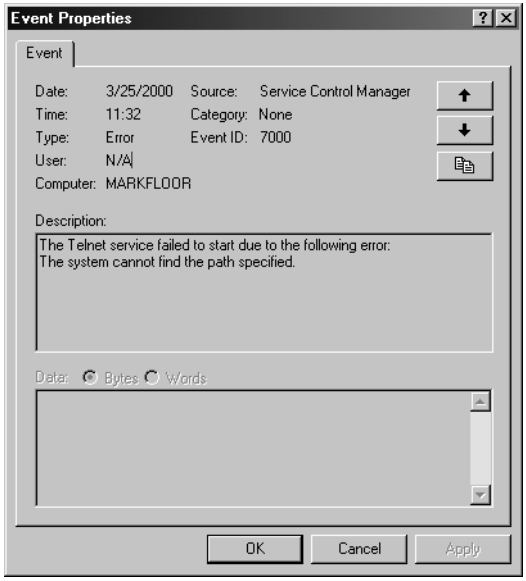


Figure 4-15 Service startup failure Event Log entry

If a service with an ErrorControl value of SERVICE_ERROR_SEVERE (2) or SERVICE_ERROR_CRITICAL (3) reports a startup error, the SCM logs a record to the Event Log and then calls the internal function *ScRevertToLastKnownGood*. This function switches the system's registry configuration to a version, named last known good, with which the system last booted successfully. Then it restarts the system using the *NtShutdownSystem* system service, which is implemented in the executive. If the system is already booting with the last known good configuration, the system just reboots.

Accepting the Boot and Last Known Good

Besides starting services, the system charges the SCM with determining when the system's registry configuration, HKLM\SYSTEM\CurrentControlSet, should be saved as the last known good control set. The CurrentControlSet key contains the Services key as a subkey, so CurrentControlSet includes the registry representation of the SCM database. It also contains the Control key, which stores many kernel-mode and user-mode subsystem configuration settings. By default, a successful boot consists of a successful startup of auto-start services and a successful user logon. A boot fails if the system halts because a device driver crashes the system during the boot or if an auto-start service with an ErrorControl value of SERVICE_ERROR_SEVERE or SERVICE_ERROR_CRITICAL reports a startup error.

The SCM obviously knows when it has completed a successful startup of the auto-start services, but Winlogon (\Windows\System32\Winlogon.exe) must notify it when there is a successful logon. Winlogon invokes the *NotifyBootConfigStatus* function when a user logs on, and *NotifyBootConfigStatus* sends a message to the SCM. Following the successful start of the auto-start services or the receipt of the message from *NotifyBootConfigStatus* (whichever comes last), the SCM calls the system function *NtInitializeRegistry* to save the current registry startup configuration.

Third-party software developers can supersede Winlogon's definition of a successful logon with their own definition. For example, a system running Microsoft SQL Server might not consider a boot successful until after SQL Server is able to accept and process transactions. Developers impose their definition of a successful boot by writing a boot-verification program and installing the program by pointing to its location on disk with the value stored in the registry key HKLM\SYSTEM\CurrentControlSet\Control\BootVerificationProgram. In addition, a boot-verification program's installation must disable Winlogon's call to *NotifyBootConfigStatus* by setting HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\ReportBootOk to 0. When a boot-verification program is installed, the SCM launches it after finishing auto-start services and waits for the program's call to *NotifyBootConfigStatus* before saving the last known good control set.

Windows maintains several copies of CurrentControlSet, and CurrentControlSet is really a symbolic registry link that points to one of the copies. The control sets have names in the form HKLM\SYSTEM\ControlSet*nnn*, where *nnn* is a number such as 001 or 002. The HKLM\SYSTEM\Select key contains values that identify the role of each control set. For example, if Cur-

rentControlSet points to ControlSet001, the Current value under Select has a value of 1. The LastKnownGood value under Select contains the number of the last known good control set, which is the control set last used to boot successfully. Another value that might be on your system under the Select key is Failed, which points to the last control set for which the boot was deemed unsuccessful and aborted in favor of an attempt at booting with the last known good control set. Figure 4-16 displays a system's control sets and Select values.

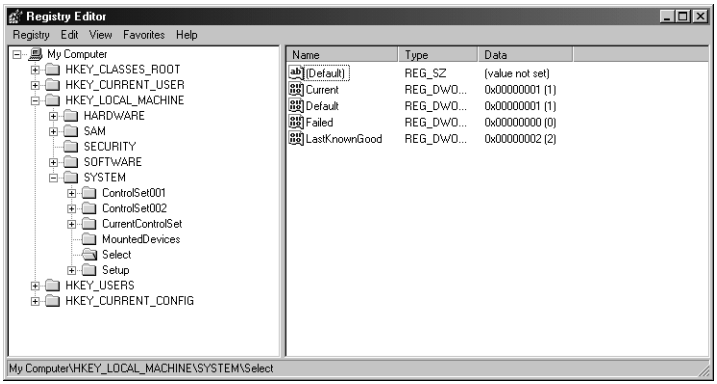


Figure 4-16 Control set selection key

NtInitializeRegistry takes the contents of the last known good control set and synchronizes it with that of the CurrentControlSet key's tree. If this was the system's first successful boot, the last known good won't exist and the system will create a new control set for it. If the last known good tree exists, the system simply updates it with differences between it and CurrentControlSet.

Last known good is helpful in situations in which a change to CurrentControlSet, such as the modification of a system performance-tuning value under HKLM\SYSTEM\Control or the addition of a service or device driver, causes the subsequent boot to fail. Users can press F8 early in the boot process to bring up a menu that lets them direct the boot to use the last known good control set, rolling the system's registry configuration back to the way it was the last time the system booted successfully. Chapter 5 describes in more detail the use of Last Known Good and other recovery mechanisms for troubleshooting system startup problems.

Service Failures

A service can have optional FailureActions and FailureCommand values in its registry key that the SCM records during the service's startup. The SCM registers with the system so that the system signals the SCM when a service process exits. When a service process terminates unexpectedly, the SCM determines which services ran in the process and takes the recovery steps specified by their failure-related registry values.

Actions that a service can configure for the SCM include restarting the service, running a program, and rebooting the computer. Furthermore, a service can specify the failure actions that

take place the first time the service process fails, the second time, and subsequent times, and it can indicate a delay period that the SCM waits before restarting the service if the service asks to be restarted. The service failure action of the IIS Admin Service results in the SCM running the IISReset application, which performs cleanup work and then restarts the service. You can easily manage the recovery actions for a service with the Recovery tab of the service's Properties dialog box in the Services MMC snap-in, as shown in Figure 4-17.

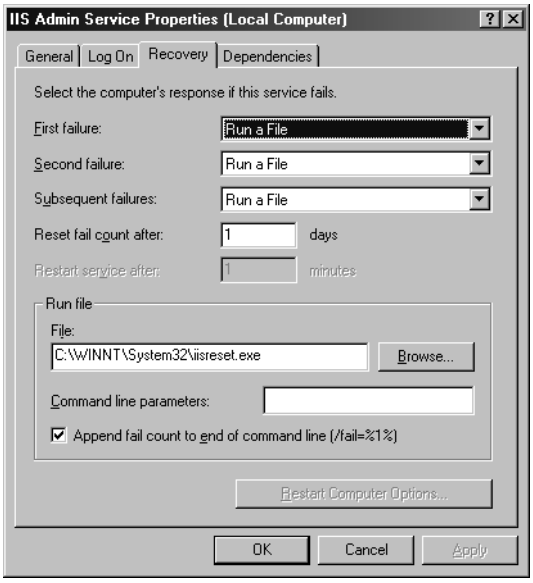


Figure 4-17 Service recovery options

Service Shutdown

When Winlogon calls the Windows *ExitWindowsEx* function, *ExitWindowsEx* sends a message to Csrss, the Windows subsystem process, to invoke Csrss's shutdown routine. Csrss loops through the active processes and notifies them that the system is shutting down. For every system process except the SCM, Csrss waits up to the number of seconds specified by HKU\.DEFAULT\Control Panel\Desktop\WaitToKillAppTimeout (which defaults to 20 seconds) for the process to exit before moving on to the next process. When Csrss encounters the SCM process, it also notifies it that the system is shutting down but employs a timeout specific to the SCM. Csrss recognizes the SCM using the process ID Csrss saved when the SCM registered with Csrss using the *RegisterServicesProcess* function during system initialization. The SCM's timeout differs from that of other processes because Csrss knows that the SCM communicates with services that need to perform cleanup when they shut down, and so an administrator might need to tune only the SCM's timeout. The SCM's timeout value resides in the HKLM\SYSTEM\CurrentControlSet\Control\WaitToKillServiceTimeout registry value, and it defaults to 20 seconds.

The SCM's shutdown handler is responsible for sending shutdown notifications to all the services that requested shutdown notification when they initialized with the SCM. The SCM function *ScShutdownAllServices* loops through the SCM services database searching for services desiring shutdown notification and sends each one a shutdown command. For each service to which it sends a shutdown command, the SCM records the value of the service's *wait hint*, a value that a service also specifies when it registers with the SCM. The SCM keeps track of the largest wait hint it receives. After sending the shutdown messages, the SCM waits either until one of the services it notified of shutdown exits or until the time specified by the largest wait hint passes.

If the wait hint expires without a service exiting, the SCM determines whether one or more of the services it was waiting on to exit have sent a message to the SCM telling the SCM that the service is progressing in its shutdown process. If at least one service made progress, the SCM waits again for the duration of the wait hint. The SCM continues executing this wait loop until either all the services have exited or none of the services upon which it's waiting has notified it of progress within the wait hint timeout period.

While the SCM is busy telling services to shut down and waiting for them to exit, Csrss waits for the SCM to exit. If Csrss's wait ends without the SCM having exited (the WaitToKillServiceTimeout time expires), Csrss simply moves on, continuing the shutdown process. Thus, services that fail to shut down in a timely manner are simply left running, along with the SCM, as the system shuts down. Unfortunately, there's no way for administrators to know whether they should raise the WaitToKillServiceTimeout value on systems where services aren't getting a chance to shut down completely before the system shuts down. See "Shutdown" in Chapter 5 for more information on the shutdown process.

Shared Service Processes

Running every service in its own process instead of having services share a process whenever possible wastes system resources. However, sharing processes means that if any of the services in the process has a bug that causes the process to exit, all the services in that process terminate.

Of the Windows built-in services, some run in their own process and some share a process with other services. For example, the SCM process hosts the Event Log service and the user-mode Plug and Play service, and the LSASS process contains security-related services—such as the Security Accounts Manager (SamSs) service, the Net Logon (Netlogon) service, and the IPSec Policy Agent (PolicyAgent) service.

There is also a "generic" process named Service Host (SvcHost - \Windows\System32\Svchost.exe) to contain multiple services. Multiple instances of SvcHost can be running in different processes. Services that run in SvcHost processes include Telephony

(TapiSrv), Remote Procedure Call (RpcSs), and Remote Access Connection Manager (RasMan). Windows implements services that run in SvcHost as DLLs and includes an ImagePath definition of the form "%SystemRoot%\System32\svchost.exe -k netsvcs" in the service's registry key. The service's registry key must also have a registry value named ServiceDll under a Parameters subkey that points to the service's DLL file.

All services that share a common SvcHost process specify the same parameter ("-k netsvcs" in the example in the preceding paragraph) so that they have a single entry in the SCM's image database. When the SCM encounters the first service that has a SvcHost ImagePath with a particular parameter during service startup, it creates a new image database entry and launches a SvcHost process with the parameter. The new SvcHost process takes the parameter and looks for a value having the same name as the parameter under HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Svchost. SvcHost reads the contents of the value, interpreting it as a list of service names, and notifies the SCM that it's hosting those services when SvcHost registers with the SCM. Figure 4-18 presents an example Svchost registry key that shows that a SvcHost process started with the "-k netsvcs" parameter is prepared to host a number of different network-related services.

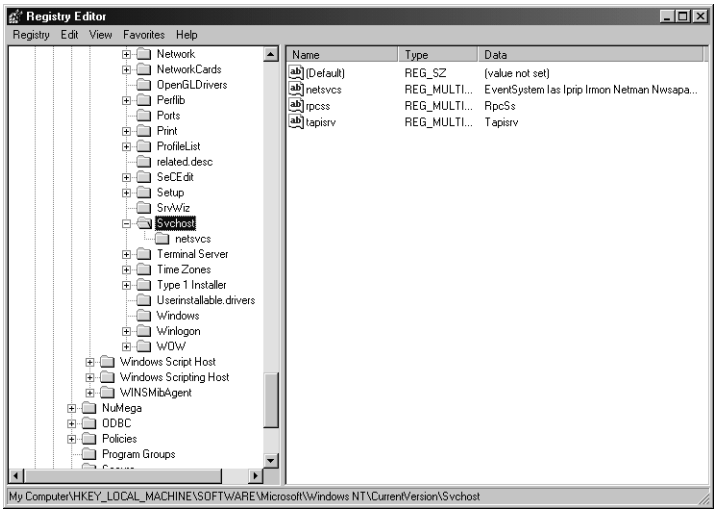
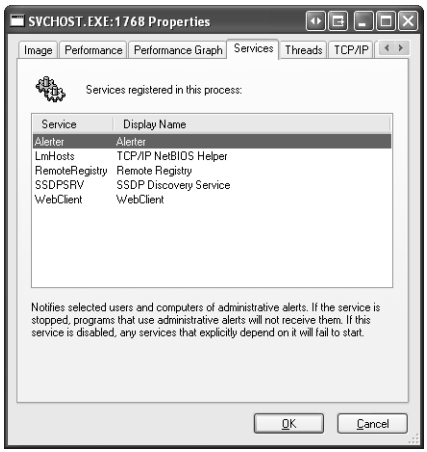


Figure 4-18 Svchost registry key

When the SCM encounters a SvcHost service during service startup with an ImagePath matching an entry it already has in the image database, it doesn't launch a second process but instead just sends a start command for the service to the SvcHost it already started for that ImagePath value. The existing SvcHost process reads the ServiceDll parameter in the service's registry key and loads the DLL into its process to start the service.

EXPERIMENT: Viewing Services Running Inside Processes

The Process Explorer utility that you can download from *www.sysinternals.com* shows detailed information about the services running with processes. Run Process Explorer and view Services tabs on the process properties dialog box for the following processes: Services.exe, Lsass.exe, and Svchost.exe. Several instances of SvcHost will be running on your system, and you can see the account in which each is running by adding the Username column to the Process Explorer display or by looking at the Username field on the Image tab of a process's Process Properties dialog box. The following figure shows the list of services running within a SvcHost executing in the local service account:



The information displayed includes the service name, display name, and service description, if it has one, which Process Explorer shows beneath the service list when you select a service.

You can also use the tlist.exe tool from the Windows Support Tools or Tasklist, which ships with Windows XP and Windows Server 2003, to view the list of services running within processes from a command prompt. The syntax to see services with Tlist is:

```
tlist /s
```

The syntax for tasklist is:

```
tasklist /svc
```

Note that these utilities do not show service display names or descriptions, only service names.

Service Control Programs

Service control programs are standard Windows applications that use XSCM service management functions, including *CreateService*, *OpenService*, *StartService*, *ControlService*, *QueryServiceStatus,* and *DeleteService*. To use the SCM functions, an SCP must first open a communications channel to the SCM by calling the *OpenSCManager* function. At the time of the open call, the SCP must specify what types of actions it wants to perform. For example, if an SCP simply wants to enumerate and display the services present in the SCM's database, it requests enumerate-service access in its call to *OpenSCManager*. During its initialization, the SCM creates an internal object that represents the SCM database and uses the Windows security functions to protect the object with a security descriptor that specifies what accounts can open the object with what access permissions. For example, the security descriptor indicates that the Authenticated Users group can open the SCM object with enumerate-service access. However, only administrators can open the object with the access required to create or delete a service.

As it does for the SCM database, the SCM implements security for services themselves. When an SCP creates a service by using the *CreateService* function, it specifies a security descriptor that the SCM associates internally with the service's entry in the service database. The SCM stores the security descriptor in the service's registry key as the Security value, and it reads that value when it scans the registry's Services key during initialization so that the security settings persist across reboots. In the same way that an SCP must specify what types of access it wants to the SCM database in its call to *OpenSCManager*, an SCP must tell the SCM what access it wants to a service in a call to *OpenService*. Accesses that an SCP can request include the ability to query a service's status and to configure, stop, and start a service.

The SCP you're probably most familiar with is the Services MMC snap-in that's included in Windows, which resides in \Windows\System32\Filemgmt.dll. Windows XP and Windows Server 2003 include Sc.exe (Service Controller tool), a command-line service control program that's available for Windows 2000 in the Windows 2000 resource kits.

SCPs sometimes layer service policy on top of what the SCM implements. A good example is the timeout that the Services MMC snap-in implements when a service is started manually. The snap-in presents a progress bar that represents the progress of a service's startup. Whereas the SCM waits indefinitely for a service to respond to a start command, the Services snap-in waits only 2 minutes before the progress bar reaches 100 percent and the snap-in announces that the service didn't start in a timely manner. Services indirectly interact with SCPs by setting their configuration status to reflect their progress as they respond to SCM commands such as the start command. SCPs query the status with the *QueryServiceStatus* function. They can tell when a service actively updates the status versus when a service appears to be hung, and the SCM can take appropriate actions in notifying a user about what the service is doing.

Windows Management Instrumentation

Windows NT has always had integrated performance and system-event monitoring tools. Applications and the system typically use the Event Manager to report errors and diagnostic messages. The Event Viewer utility lets administrators view event output from either the local computer or another computer on the network. Similarly, the performance counter mechanism lets applications and operating system components report performance-related statistics to performance-monitoring applications such as the Performance Monitor.

Although the Windows NT 4 event-monitoring and performance-monitoring features met their design goals, they had limitations. For example, the programming interfaces differ from one another, and this variation increases the complexity of applications that use both event and performance monitoring to collect data. Perhaps the biggest drawback to the monitoring facilities in Windows NT 4 is that they have little or no extensibility and that neither event logging nor performance data collection provides the two-way interaction necessary in a management API. Applications must provide data in predefined formats. The Performance API provides no way for an application to receive notification of performance-related events, and applications that request notification of Event Manager events can't restrict notification to specific event types or sources. Finally, clients of either collection facility can't communicate with event-data or performance-data providers through the Event Manager or Performance API.

To address these limitations as well as to provide management capabilities for other types of data sources, Windows has a new management mechanism, Windows Management Instrumentation (WMI). WMI is an implementation of Web-Based Enterprise Management (WBEM), a standard that the Distributed Management Task Force (DMTF—an industry consortium) defines. The WBEM standard encompasses the design of an extensible enterprise data-collection and data-management facility that has the flexibility and extensibility required to manage local and remote systems that comprise arbitrary components. WMI support was added to Windows NT 4 in Service Pack 4. It is also supported in Windows 95 OSR2, Windows 98 and Windows Millennium. Although most of this section applies to all the Windows platforms that support WMI, implementation details are specific to Windows 2000, Windows XP, and Windows Server 2003.

WMI Architecture

WMI consists of four main components, as shown in Figure 4-19: management applications, WMI infrastructure, providers, and managed objects. Management applications are Windows applications that access and display or process data that the applications obtain about managed objects. A simple example of a management application is a Performance tool replacement that relies on WMI rather than the Performance API to obtain performance information. A more complex example is an enterprise-management tool that lets administrators perform automated inventories of the software and hardware configuration of every computer in their enterprise.

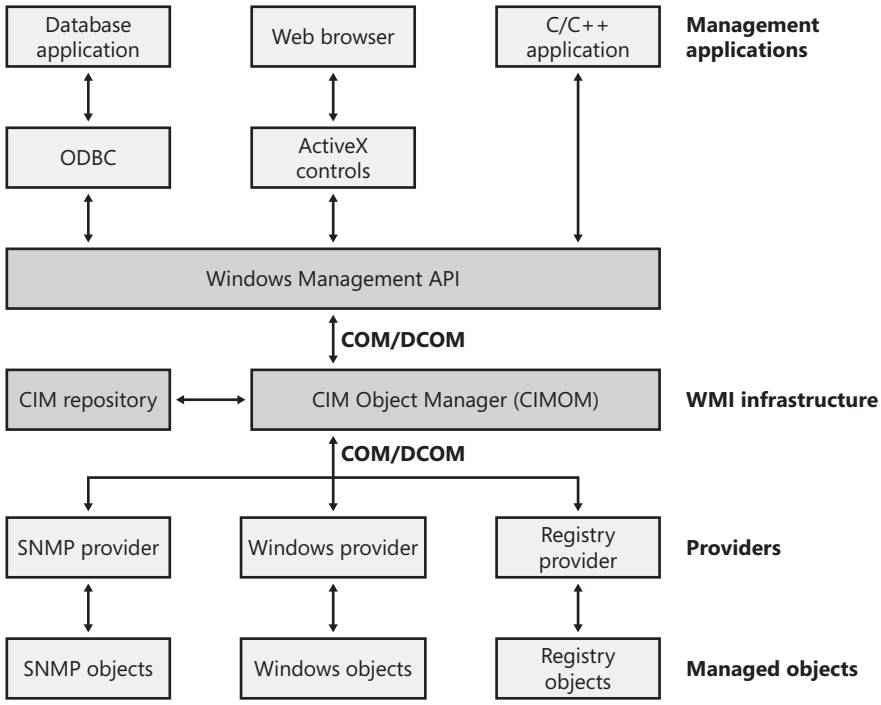


Figure 4-19 WMI architecture

Developers typically must target management applications to collect data from and manage specific objects. An object might represent one component, such as a network adapter device, or a collection of components, such as a computer. (The computer object might contain the network adapter object.) Providers need to define and export the representation of the objects that management applications are interested in. For example, the vendor of a network adapter might want to add adapter-specific properties to the network adapter WMI support that Windows includes, querying and setting the adapter's state and behavior as the management applications direct. In some cases (for example, for device drivers), Microsoft supplies a provider that has its own API to help developers leverage the provider's implementation for their own managed objects with minimal coding effort.

The WMI infrastructure, the heart of which is the Common Information Model (CIM) Object Manager (CIMOM), is the glue that binds management applications and providers. (CIM is described later in this chapter.) The infrastructure also serves as the object-class store and, in many cases, as the storage manager for persistent object properties. WMI implements the store, or repository, as an on-disk database named the CIMOM Object Repository. As part of its infrastructure, WMI supports several APIs through which management applications access object data and providers supply data and class definitions.

Windows programs use the WMI COM API, the primary management API, to directly interact with WMI. Other APIs layer on top of the COM API and include an Open Database Connectivity (ODBC) adapter for the Microsoft Access database application. A database developer uses the WMI ODBC adapter to embed references to object data in the developer's database. Then the developer can easily generate reports with database queries that contain WMI-based data. WMI ActiveX controls support another layered API. Web developers use the ActiveX controls to construct Web-based interfaces to WMI data. Another management API is the WMI scripting API, for use in script-based applications and Microsoft Visual Basic programs. WMI scripting support exists for all Microsoft programming language technologies.

As they are for management applications, WMI COM interfaces constitute the primary API for providers. However, unlike management applications, which are COM clients, providers are COM or Distributed COM (DCOM) servers (that is, the providers implement COM objects that WMI interacts with). Possible embodiments of a WMI provider include DLLs that load into WMI's manager process or stand-alone Windows applications or Windows services. Microsoft includes a number of built-in providers that present data from well-known sources, such as the Performance API, the registry, the Event Manager, Active Directory, SNMP, and Windows Driver Model (WDM) device drivers. The WMI SDK lets developers develop third-party WMI providers.

Providers

At the core of WBEM is the DMTF-designed CIM specification. The CIM specifies how management systems represent, from a systems management perspective, anything from a computer to an application or device on a computer. Provider developers use the CIM to represent the components that make up the parts of an application for which the developers want to enable management. Developers use the Managed Object Format (MOF) language to implement a CIM representation.

In addition to defining classes that represent objects, a provider must interface WMI to the objects. WMI classifies providers according to the interface features the providers supply. Table 4-10 lists WMI provider classifications. Note that a provider can implement one or more features; therefore, a provider can be, for example, both a class and an event provider. To clarify the feature definitions in Table 4-10, let's look at a provider that implements several of those features. The Event Log provider supports several objects, including an Event Log Computer, an Event Log Record, and an Event Log File. The Event Log is an Instance provider because it can define multiple instances for several of its classes. One class for which the Event Log provider defines multiple instances is the Event Log File class (Win32_NTEventlogFile); the Event Log provider defines an instance of this class for each of the system's event logs (that is, System Event Log, Application Event Log, and Security Event Log).

Table 4-10 Provider Classifications

Classification	Description
Class	Can supply, modify, delete, and enumerate a provider-specific class. Can also support query processing. Active Directory is a rare example of a service that is a class provider.
Instance	Can supply, modify, delete, and enumerate instances of system and provider-specific classes. An instance represents a managed object. Can also support query processing.
Property	Can supply and modify individual object property values.
Method	Supplies methods for a provider-specific class.
Event	Generates event notifications.
Event consumer	Maps a physical consumer to a logical consumer to support event notification.

The Event Log provider defines the instance data and lets management applications enumerate the records. To let management applications use WMI to back up and restore the Event Log files, the Event Log provider implements backup and restore methods for Event Log File objects. Doing so makes the Event Log provider a Method provider. Finally, a management application can register to receive notification whenever a new record writes to one of the Event Logs. Thus, the Event Log provider serves as an Event provider when it uses WMI event notification to tell WMI that Event Log records have arrived.

The Common Information Model and the Managed Object Format Language

The CIM follows in the steps of object-oriented languages such as C++ and Java, in which a modeler designs representations as classes. Working with classes lets developers use the powerful modeling techniques of inheritance and composition. Subclasses can inherit the attributes of a parent class, and they can add their own characteristics and override the characteristics they inherit from the parent class. A class that inherits properties from another class *derives* from that class. Classes also *compose*: a developer can build a class that includes other classes.

The DMTF provides multiple classes as part of the WBEM standard. These classes are CIM's basic language and represent objects that apply to all areas of management. The classes are part of the CIM *core model*. An example of a core class is CIM_ManagedSystemElement. This class contains a few basic properties that identify physical components such as hardware devices, and logical components such as processes and files. The properties include a caption, description, installation date, and status. Thus, the CIM_LogicalElement and CIM_PhysicalElement classes inherit the attributes of the CIM_ManagedSystemElement class. These two classes are also part of the CIM core model. The WBEM standard calls these classes *abstract* classes because they exist solely as classes that other classes inherit (that is, no

object instances of an abstract class exist). You can therefore think of abstract classes as templates that define properties for use in other classes.

A second category of classes represents objects that are specific to management areas but independent of a particular implementation. These classes constitute the *common model* and are considered an extension of the core model. An example of a common-model class is the CIM_FileSystem class, which inherits the attributes of CIM_LogicalElement. Because virtually every operating system—including Windows, Linux, and other varieties of UNIX—rely on file-system-based structured storage, the CIM_FileSystem class is an appropriate constituent of the common model.

The final class category, the *extended model*, comprises technology-specific additions to the common model. Windows defines a large set of these classes to represent objects specific to the Windows environment. Because all operating systems store data in files, the CIM common model includes the CIM_LogicalFile class. The CIM_DataFile class inherits the CIM_LogicalFile class, and Windows adds the Win32_PageFile and Win32_ShortcutFile file classes for those Windows file types.

The Event Log provider makes extensive use of inheritance. Figure 4-20 shows a view of the WMI CIM Studio, a class browser that ships with the WMI Administrative Tools that you can obtain from the Microsoft download center at the Microsoft Web site. You can see where the Event Log provider relies on inheritance in the provider's Win32_NTEventlogFile class, which derives from CIM_DataFile. Event Log files are data files that have additional Event Log–specific attributes such as a log file name (LogfileName) and a count of the number of records that the file contains (NumberOfRecords). The tree that the class browser shows reveals that Win32_NTEventlogFile is based on several levels of inheritance, in which CIM_DataFile derives from CIM_LogicalFile, which derives from CIM_LogicalElement, and CIM_LogicalElement derives from CIM_ManagedSystemElement.

As stated earlier, WMI provider developers write their classes in the MOF language. The following output shows the definition of the Event Log provider's Win32_NTEventlogFile, which is selected in Figure 4-20. Notice the correlation between the properties that the right panel in Figure 4-20 lists and those properties' definitions in the MOF file below. CIM Studio uses yellow arrows to tag those properties that a class inherits. Thus, you don't see those properties specified in Win32_NTEventlogFile's definition.

```
dynamic: ToInstance, provider("MS_NT_EVENTLOG_PROVIDER"), Locale(1033), UUID("{8502C57B-
5FBB-11D2-AAC1-006008C78BC7}")]
class Win32_NTEventlogFile : CIM_DataFile
{
[read] string LogfileName;
[read, write] uint32 MaxFileSize;
[read] uint32 NumberOfRecords;
[read, volatile, ValueMap{"0", "1..365", "4294967295"}] string OverWritePolicy;
[read, write, Units("Days"), Range("0-365 | 4294967295")] uint32 OverwriteOutDated;
[read] string Sources[];
```

```
[implemented, Privileges{"SeSecurityPrivilege",
"SeBackupPrivilege"}] uint32 ClearEventlog([in] string ArchiveFileName);
[implemented, Privileges{"SeSecurityPrivilege",
"SeBackupPrivilege"}] uint32 BackupEventlog([in] string ArchiveFileName);
};
```

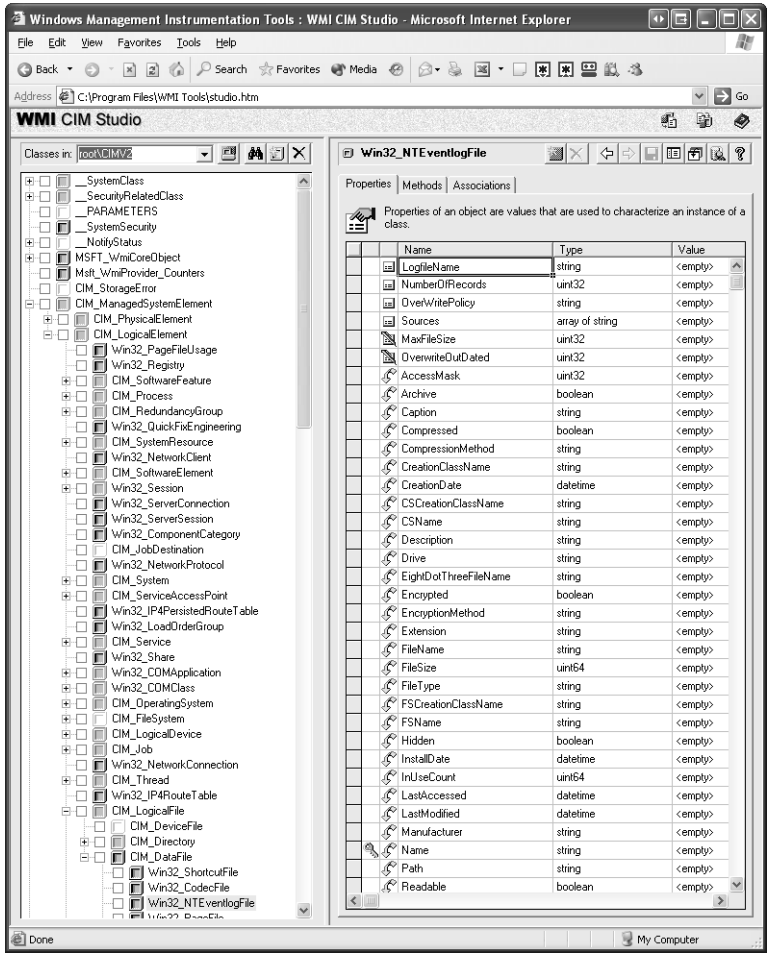


Figure 4-20 WMI CIM Studio

One term worth reviewing is *dynamic*, which is a descriptive designator for the Win32_NTEventlogFile class that the MOF file in the preceding output shows. *Dynamic* means that the WMI infrastructure asks the WMI provider for the values of properties associated with an object of that class whenever a management application queries the object's properties. A *static* class is one in the WMI repository; the WMI infrastructure refers to the repository to obtain the values instead of asking a provider for the values. Because updating the repository is a relatively expensive operation, dynamic providers are more efficient for objects that have properties that change frequently.

EXPERIMENT: Viewing the MOF Definitions of WMI Classes

You can view the MOF definition for any WMI class by using the WbemTest tool that comes with Windows. In this experiment, we'll look at the MOF definition for the Win32_NTEventLogFile class:

1. Run Wbemtest from the Start menu's Run dialog box.
2. Click the Connect button, change the Namespace to root\cimv2, and connect.
3. Select Enum Classes, select the Recursive option button, and then click OK.
4. Find Win32_NTEventLogFile in the list classes, and double-click it to see its class properties.
5. Click the Show MOF button to open a window that displays the MOF text.

After constructing classes in MOF, WMI developers can supply the class definitions to WMI in several ways. WDM provider developers compile a MOF file into a binary MOF (BMF) file—a more compact binary representation than a MOF file—and give the BMF files to the WDM infrastructure. Another way is for the provider to compile the MOF and use WMI COM APIs to give the definitions to the WMI infrastructure. Finally, a provider can use the MOF Compiler (Mofcomp.exe) tool to give the WMI infrastructure a classes-compiled representation directly.

The WMI Namespace

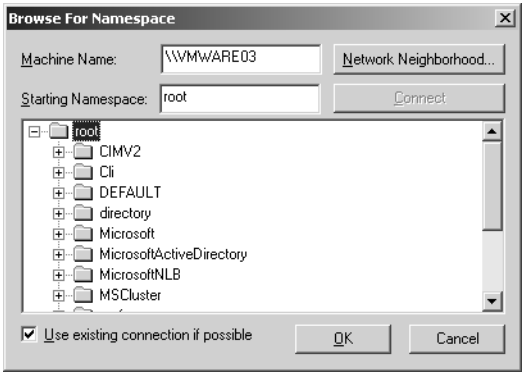
Classes define the properties of objects, and objects are class instances on a system. WMI uses a namespace that contains several subnamespaces that WMI arranges hierarchically to organize objects. A management application must connect to a namespace before the application can access objects within the namespace.

WMI names the namespace root directory *root*. All WMI installations have four predefined namespaces that reside beneath root: CIMV2, Default, Security, and WMI. Some of these namespaces have other namespaces within them. For example, CIMV2 includes the Applications and ms_409 namespaces as subnamespaces. Providers sometimes define their own namespaces; you can see the WMI namespace (which the Windows device driver WMI provider defines) beneath root in Windows.

EXPERIMENT: Viewing WMI Namespaces

You can see what namespaces are defined on a system with WMI CIM Studio. WMI CIM Studio presents a connection dialog box when you run it that includes a namespace browsing button to the right of the namespace edit box. Opening the browser and selecting a namespace has WMI CIM Studio connect to that namespace.

Windows Server 2003 defines over a dozen namespaces beneath root, some of which are visible here:



Unlike a file system namespace, which comprises a hierarchy of directories and files, a WMI namespace is only one level deep. Instead of using names as a file system does, WMI uses object properties that it defines as *keys* to identify the objects. Management applications specify class names with key names to locate specific objects within a namespace. Thus, each instance of a class must be uniquely identifiable by its key values. For example, the Event Log provider uses the Win32_NTLogEvent class to represent records in an Event Log. This class has two keys: Logfile, a string; and RecordNumber, an unsigned integer. A management application that queries WMI for instances of Event Log records obtains them from the provider key pairs that identify records. The application refers to a record using the syntax that you see in this sample object pathname:

```
\\DARYL\root\CIMV2:Win32_NTLogEvent.Logfile="Application",
                                    RecordNumber="1"
```

The first component in the name (\\DARYL) identifies the computer on which the object is located, and the second component (\root\CIMV2) is the namespace in which the object resides. The class name follows the colon, and key names and their associated values follow the period. A comma separates the key values.

WMI provides interfaces that let applications enumerate all the objects in a particular class or to make queries that return instances of a class that match a query criteria.

Class Association

Many object types are related to one another in some way. For example, a computer object has a processor, software, an operating system, active processes, and so on. WMI lets providers construct an *association class* to represent a logical connection between two different classes. Association classes associate one class with another, so the classes have only two properties: a class name and the *Ref* modifier. The following output shows an association in which the

Event Log provider's MOF file associates the Win32_NTLogEvent class with the Win32_ComputerSystem class. Given an object, a management application can query associated objects. In this way, a provider defines a hierarchy of objects.

```
[dynamic: ToInstance, provider("MS_NT_EVENTLOG_PROVIDER"): ToInstance, EnumPrivileges{"SeSec
urityPrivilege"}: ToSubClass, Locale(1033): ToInstance, UUID("{8502C57F-5FBB-11D2-AAC1-
006008C78BC7}"): ToInstance, Association: DisableOverride ToInstance ToSubClass]
class Win32_NTLogEventComputer
{
    [key, read: ToSubClass] Win32_ComputerSystem ref Computer;
    [key, read: ToSubClass] Win32_NTLogEvent ref Record;
};
```

Figure 4-21 shows the WMI Object Browser (another tool that the WMI Administrative Tools includes) displaying the contents of the CIMV2 namespace. Windows system components typically place their objects within the CIMV2 namespace. The Object Browser first locates the Win32_ComputerSystem object instance MR-XEON, which is the object that represents the computer. Then the Object Browser obtains the objects associated with Win32_ComputerSystem and displays them beneath MR-XEON. The Object Browser user interface displays association objects with a double-arrow folder icon. The associated class type's objects display beneath the folder.

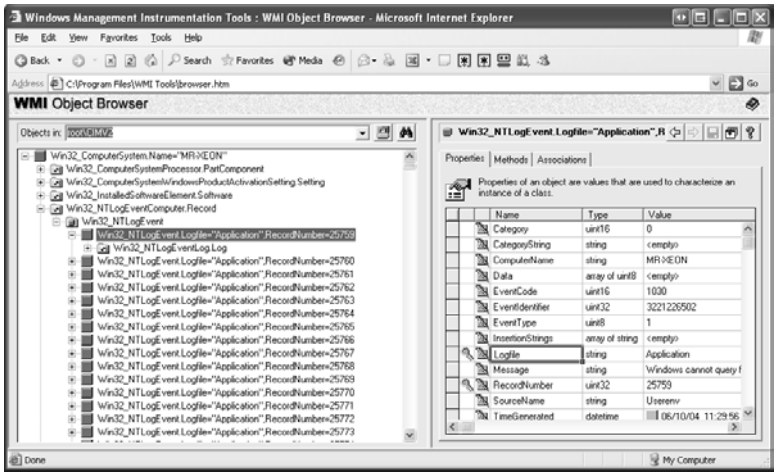


Figure 4-21 WMI Object Browser

You can see in the Object Browser that the Event Log provider's association class Win32_NTLogEventComputer is beneath MR-XEON and that numerous instances of the Win32_NTLogEvent class exist. Refer to the preceding output to verify that the MOF file defines the Win32_NTLogEventComputer class to associate the Win32_ComputerSystem class with the Win32_NTLogEvent class. Selecting an instance of Win32_NTLogEvent in the Object Browser reveals that class's properties under the Properties tab in the right-hand pane. Microsoft intended the Object Browser to help WMI developers examine their objects, but a management application would perform the same operations and display properties or collected information more intelligibly.

EXPERIMENT: Using WMI Scripts to Manage Systems

A powerful aspect of WMI is its support for scripting languages. Microsoft has generated hundreds of scripts that perform common administrative tasks for managing user accounts, files, the registry, processes, and hardware devices. While some scripts ship in the Windows Resource Kits, the Microsoft TechNet Scripting Center Web site serves as the central location for Microsoft scripts. Using a script from the scripting center is as easy as copying its text from your Internet browser, storing it in a file with with a .vbs extension, and running it with the command cscript **script**.vbs, where "script" is the name you gave the script. Cscript is the command-line interface to Windows Script Host (WSH).

Here's a sample TechNet script that registers to receive events when Win32_Process object instances are created, which occurs whenever a process starts, and prints a line with the name of the process that the object represents:

```
strComputer = "."
Set objWMIService = GetObject("winmgmts:" _
    & "{impersonationLevel=impersonate}!\\" & strComputer & "\root\cimv2")
Set colMonitoredProcesses = objWMIService. _
    ExecNotificationQuery("select * from __instancecreationevent " _
        & " within 1 where TargetInstance isa 'Win32_Process'")
i = 0
Do While i = 0
    Set objLatestProcess = colMonitoredProcesses.NextEvent
    Wscript.Echo objLatestProcess.TargetInstance.Name
Loop
```

The line that invokes *ExecNotificationQuery* does so with a parameter that includes a "select" statement, which highlights WMI's support for a read-only subset of the ANSI standard Structured Query Language (SQL), known as WQL, to provide a flexible way for WMI consumers to specify the information that they want to extract from WMI providers. Running the sample script with Cscript and then starting Notepad results in the following output:

```
C:\>cscript monproc.vbs
Microsoft (R) Windows Script Host Version 5.6
Copyright (C) Microsoft Corporation 1996-2001. All rights reserved.

NOTEPAD.EXE
```

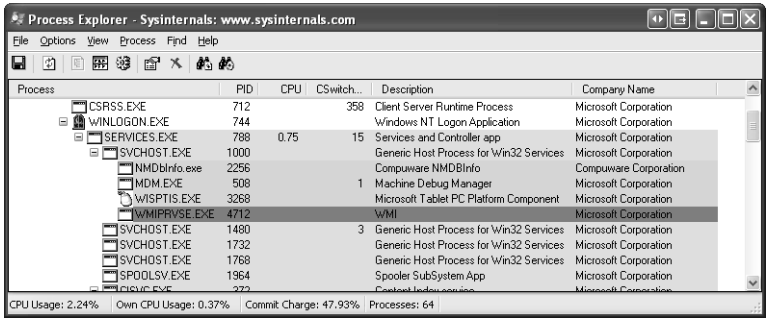
WMI Implementation

In Windows 2000, the WMI service is implemented in \Windows\System32\Winmgmt.exe, which the Windows SCM starts the first time a management application or WMI provider tries to access WMI APIs. In Windows XP and Windows Server 2003, the WMI service runs in a shared Svchost process that executes in the local system account.

In Windows 2000, WMI loads providers as in-process DCOM servers that execute within the Winmgmt service process. If a provider bug crashes the WMI process, the WMI service exits and then restarts in response to the next WMI request. Because the WMI service shares its Svchost process with several other services that would also exit if a WMI provider bug caused the process to exit, in Windows XP and Windows Server 2003, WMI loads providers into the Wmiprvse.exe provider-hosting process. Wmiprvse.exe launches as a child of the RPC service process. WMI executes Wmiprvse in the local system, local service, or network service accounts, depending on the value of the *HostingModel* property of the WMI Win32Provider object instance that represents the provider implementation. A Wmiprvse process exits after the provider is removed from the cache, one minute following the last provider request it receives.

EXPERIMENT: Viewing Wmiprvse Creation

You can see Wmiprvse being created by running Process Explorer from *www.sysinternals.com* and executing Wmic. A Wmiprvse process will appear beneath the Svchost process that hosts the RPC service. If Process Explorer job highlighting is enabled, it will appear with the job highlight color because, to prevent a runaway provider from consuming all virtual memory resources on a system, Wmiprvse executes in a job object that limits the number of child processes it can create and the amount of virtual memory each process and all the processes of the job can allocate. (See Chapter 6 for more information on job objects.)



Most WMI components reside by default in \Windows\System32 and \Windows\System32\Wbem, including Windows MOF files, built-in provider DLLs, and management application WMI DLLs. Look in the \Windows\System32\Wbem directory, and you'll find Ntevt.mof, the Event Log provider MOF file. You'll also find Ntevt.dll, the Event Log provider's DLL, which the WMI service uses.

Directories beneath \Windows\System32\Wbem store the repository, log files, and third-party MOF files. WMI implements the repository—named the CIMOM repository—using a proprietary version of the Microsoft JET database engine. In Windows 2000, the database file stores in \Windows\System32\Wbem\Repository\Cim.rep; in Windows XP and Windows Server 2003, the database resides in \Windows\System32\Wbem\Repository\Fs.

WMI honors numerous registry settings (including various internal performance parameters such as CIMOM backup locations and intervals in Windows 2000) that the service's HKLM\SOFTWARE\Microsoft\WBEM\CIMOM registry key stores.

Device drivers use special interfaces to provide data to and accept commands—called the WMI System Control commands—from WMI. These interfaces are part of the WDM, which is explained in Chapter 9. Because the interfaces are cross-platform, they fall under the \root\WMI namespace.

WMIC

Windows XP and Windows Server 2003 include Wmic.exe, a utility that allows you to interact with WMI from a WMI-aware command-line shell. All WMI objects and their properties, including their methods, are accessible through the shell, which makes WMIC an advanced systems management console.

WMI Security

WMI implements security at the namespace level. If a management application successfully connects to a namespace, the application can view and access the properties of all the objects in that namespace. An administrator can use the WMI Control application to control which users can access a namespace. To start the WMI Control application, from the Start menu, select Programs, Administrative Tools, Computer Management. Next, open the Services And Applications branch. Right-click WMI Control, and select Properties to launch the WMI Control Properties dialog box, which Figure 4-22 shows. To configure security for namespaces, click the Security tab, select the namespace, and click Security. The other tabs in the WMI Control Properties dialog box let you modify the performance and backup settings that the registry stores.

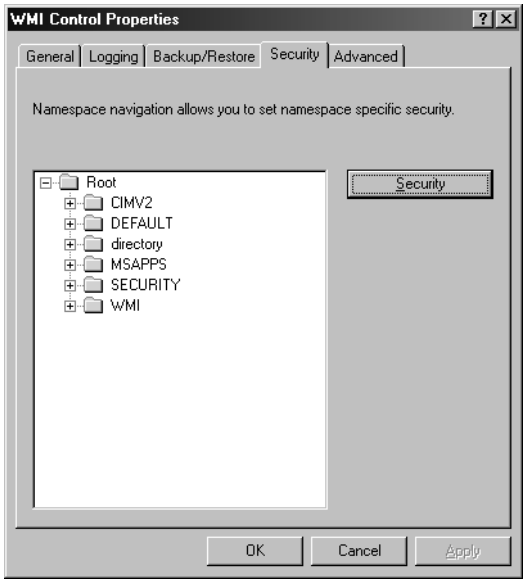


Figure 4-22 WMI security properties

Conclusion

So far, we've examined the overall structure of Windows and the core system mechanisms on which the structure is built, and core management mechanisms. With this foundation laid, we're ready to explore the boot process and the individual executive components in more detail.