Inside Win32 Services

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

Most applications are interactive in nature; that is, a user provides input, and the application provides output in response. Users start an application when they want to begin using the application's functionality, then exit the application when they're finished. However, many applications don't provide functionality directly to users. Instead, those applications manage resources that other applications or multiple users share, or provide functionality that must always be available regardless of who uses the computer. These types of applications are essentially extensions to an OS because they execute from system boot to shutdown, similarly to an OS's built-in resource management. Another characteristic of these applications is that they almost never have a user interface (UI). Because their execution usually doesn't depend on a user even being logged on to the system, these applications perform their work quietly.

In Windows 2000 (Win2K) and Windows NT, such applications are called services, or Win32 services, because they rely on the Win32 API to interact with Windows. Services resemble UNIX daemon processes and often implement the server side of client/server applications. For example, most Web servers for Win2K use a service process as the primary interface to Web clients. A Web server's requirements match the facilities that Win32 services provide: The server must be running regardless of whether a user is logged on to the computer (usually, administrators performing maintenance tasks are the only users who ever log on to servers); the server must start running when the system starts so that an administrator doesn't have to remember (or even be present at the machine) to start it; and finally, the core of a Web server requires no UI for processing Web requests. Web servers often require configuration interfaces, but separate, administrator-invoked applications that communicate settings to the service implement these interfaces.

Win32 services require three components: a service application, a service control program (SCP), and a Service Control Manager (SCM, pronounced scum). If you understand how services, SCPs, and the SCM work, you'll also have a clearer idea of what Win2K and NT are doing as the system boots and shuts down. This understanding will help you properly configure service parameters and troubleshoot problems that occur during these phases of the system life cycle. In this two-part series, you'll learn about the three Win32 services components' internals, including differences between Win2K and NT 4.0. This month, I describe service applications and service accounts, and I begin describing the SCM's operation.

Service Applications

Service applications, such as Web servers, consist of at least one executable file that runs as a Win32 service. A user who wants to start, stop, or configure a service uses an SCP. Win2K and NT both supply built-in SCPs that provide general stop, pause, and continue functionality, but applications often include their own SCP that lets an administrator specify settings particular to the service that the application manages. Microsoft supplies the SCM that starts services according to their start parameters and that serves as the communications interface for service control commands (e.g., stop, start, pause) that pass from SCPs to services.

Because services play a different role from the role that Win32 GUI applications play, you might assume that a service's implementation is different from that of an application. However, the only difference between most services and GUI applications is that a service includes code to receive commands from the SCM and to communicate the service's status back to the SCM. Because typical services don't require a UI, developers implement them as console programs. Graphical applications receive windowing event notifications and user input via their windows' message loop, and console programs that a user launches from the Run dialog box or a command prompt receive keyboard input via a command-prompt window. However, console programs that launch as services aren't visible and—most of the time—don't receive input. All Win32 executable files are formatted according to

Microsoft's Portable Executable (PE) file format, and Win2K's image loader checks flags in the image's PE header to determine whether an executable image is graphical or console-based.

When you install an application that includes a service, the application's setup program must register the service with the system. To do so, the application calls the Win32 CreateService API, a services-related API that the system implements in the Advanced API DLL, ADVAPI32.DLL (winnt\system32\advapi32.dll). ADVAPI32 implements all of the client-side SCM APIs. When a setup program registers a service, the program indirectly creates a Registry key for the service under HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services. A service's Registry key is the nonvolatile storage for a service's parameters, and so the Services key is the nonvolatile representation of the SCM's service database. The SCM database is also known as the ServicesActive database. After creating a service, an installation application can start the service with the StartService API, thus avoiding the necessity of a reboot. Many service-based applications must initialize during the boot to function, so setup programs often register a service as an automatic-start service, ask the user to reboot the system to complete the installation, and let the SCM start the service as the system boots.

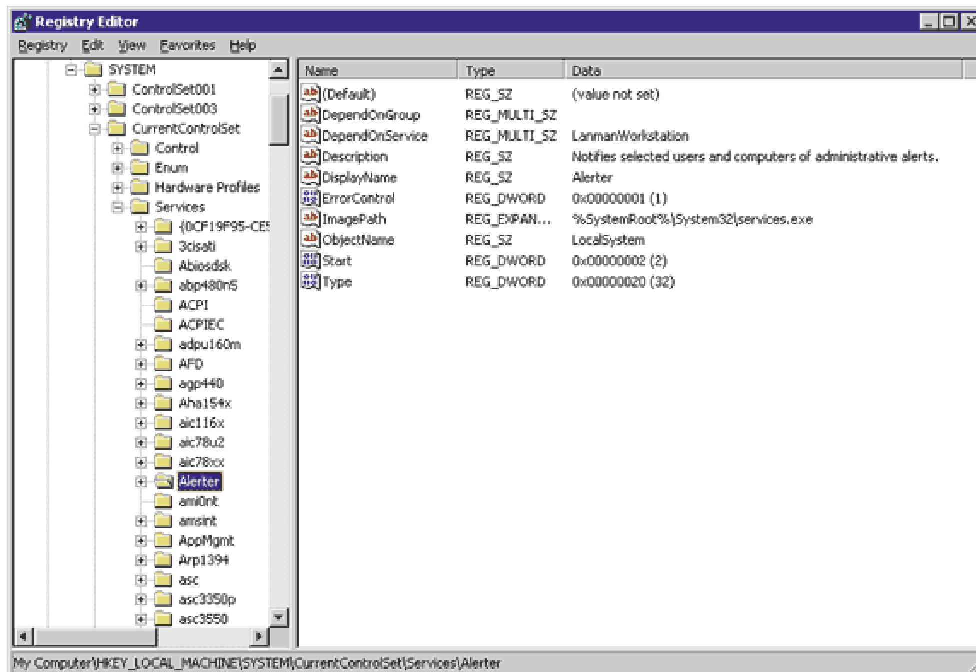
When a program calls CreateService, the program must specify the parameters that describe the service's characteristics. The characteristics include the service's type, the location of the service image file, an optional description of the service, whether the service starts automatically when the system boots or starts manually under an SCP's direction, how the system reacts if the service indicates an error when starting, and the service's group membership (i.e., optional information that specifies when the service starts with respect to other services). The SCM stores each characteristic as a value in the service's Registry key. Table 1 lists service and driver parameters, and Screen 1 shows an example of a service's Registry key. I'll describe most of the Type value's characteristics when I describe the SCM; however, the Type characteristic in general affects a service's implementation and requires explanation here.

TABLE 1: Service and Driver Registry Parameters

Value Name	Value Setting	Value Setting Description
DependOnGroup	Group Name	Drivers or services won't load unless a driver or service from the specified group loads.
DependOnService	Service Name	Services won't load until after the specified service loads. This parameter doesn't apply to device drivers.
Description	Description of service	Up to 1024-byte description of the service. New to Win2K.
DisplayName	Name of service	The Services application shows services by this name. If no name is specified, then the name of the service's Registry key serves as the service's name.
ErrorControl	IGNORE (0)	The I/O Manager ignores errors the driver returns. No warning logs or displays.
	NORMAL (1)	If the driver reports an error, a

		warning displays.
	SEVERE (2)	If the driver returns an error and the user doesn't use LastKnownGood, reboot into LastKnownGood. Otherwise, continue the boot.
	CRITICAL (3)	If the driver returns an error and the user doesn't use LastKnownGood, reboot into LastKnownGood. Otherwise, stop the boot with a blue screen crash.
FailureActions	Description of actions the SCM should take when the 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. New to Win2K.
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. New to Win2K.
Group	Group Name	A driver or service that initializes when its group initializes.
ImagePath	Path to service or driver executable file	If ImagePath isn't specified, the I/O Manager looks for drivers in \winnt\system32\drivers, and the SCM looks for services in \winnt\system32.
ObjectName	Usually LocalSystem, but can be account name such as Administrator	Specifies the account in which the service will run. If Object-Name isn't specified, LocalSystem is used. This parameter doesn't apply to device drivers.
Start	SERVICE_BOOT_START (0)	NT Loader (NTLDR) or OSLOADER preloads the driver so that the driver is in memory during the boot. SERVICE_BOOT_START drivers initialize just before SERVICE_SYSTEM_START drivers.
	SERVICE_SYSTEM_START (1)	The driver loads and initializes after SERVICE_BOOT_START drivers have initialized.
	SERVICE_AUTO_START (2)	The SCM starts the driver or service.
	SERVICE_DEMAND_START (3)	The SCM must start the driver or service on demand.
	SERVICE_DISABLED (4)	The driver or service doesn't load or initialize.
Tag	Tag Number	A specified location in a group initialization order. This parameter

		doesn't apply to services.
Type	SERVICE_KERNEL_DRIVER (1)	Device driver.
	SERVICE_FILE_SYSTEM_DRIVER (2)	Kernel-mode file-system driver.
	SERVICE_RECOGNIZER_DRIVER (8)	File-system recognizer driver.
	SERVICE_WIN32_OWN_PROCESS (16)	A service that runs in a process that hosts only one service.
	SERVICE_WIN32_SHARE_PROCESS (32)	A service that runs in a process that hosts multiple services.
	SERVICE_INTERACTIVE_PROCESS (256)	A service that can display windows on the console and receive user input.



The Type values that Table 1 shows include three that apply to device drivers:

- SERVICE_KERNEL_DRIVER
- SERVICE_FILE_SYSTEM_DRIVER
- SERVICE_RECOGNIZER_DRIVER

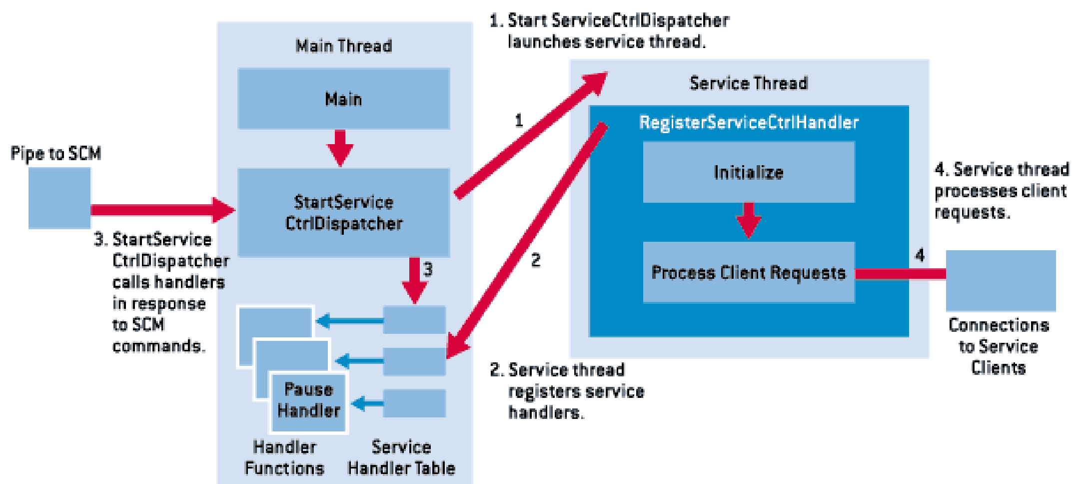
Win2K device drivers use these values and store their parameters as Registry data in the Services Registry branch. The SCM starts drivers with auto-start Start values, so the services database naturally includes drivers. Services use other Type values, SERVICE_WIN32_OWN_PROCESS and SERVICE_WIN32_SHARE_PROCESS, which are mutually exclusive. An executable file can host more than one service, and executable files that do host more than one service specify the SERVICE_WIN32_SHARE_PROCESS Type value. An advantage to having a process run more than one service is that doing so saves the system resources that would otherwise be required to run the services in discrete processes. A potential disadvantage of this capability is that if one service of

multiple services that run in the same process causes an error that terminates the process, all the services in that process terminate.

When the SCM starts a service process, the process immediately invokes the StartServiceCtrlDispatcher API, which ADVAPI32 implements. StartServiceCtrlDispatcher takes a list of entry points that consists of one entry point for each service in the process; the name of the service that the entry point corresponds to identifies each entry point. After making a named-pipe communications connection to the SCM, StartServiceCtrlDispatcher waits in a loop for commands to come through the pipe from the SCM. The SCM sends a service-start command each time it starts a service that the process owns. For each start command it receives, the StartServiceCtrlDispatcher function creates 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' main function only when all of the process' service threads have terminated. This behavior lets the service process clean up resources before exiting.

A service entry point's first action is to call the RegisterServiceCtrlHandler API. This routine, also implemented in ADVAPI32, receives and stores a table of functions that the service implements to handle various commands it receives from the SCM. RegisterServiceCtrlHandler doesn't communicate with the SCM but stores the table in local process memory for the StartServiceCtrlDispatcher function. The service entry point continues initializing the service, which might involve allocating memory, creating communications end points, and reading private configuration data from the Registry. A convention that most services follow is to store their parameters under the Parameters subkey of their service Registry key. While the entry point is initializing, it might periodically send the SCM status messages indicating how startup is progressing. After the entry point finishes initialization, a service thread usually waits in a loop 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' main thread, which executes in the StartServiceCtrlDispatcher function, receives SCM commands directed at services in the process and uses the service's table of handler functions to locate and invoke the service function that responds to each command. SCM commands include stop, pause, resume, interrogate, and shut down. Some commands can be application-defined commands. Figure 1, page 68, shows the internal organization of a service process. The figure depicts the two threads—the main thread and the service thread—that make up a process that hosts one service.



Service Accounts

Unless a service installation program or administrator specifies otherwise, services run in Local System, the security context of the system account. From a security perspective, this account is extremely powerful. All of Win2K's user-mode OS components run in Local System, including the Win2K session manager (\winnt\system32\smss.exe), the Win32 subsystem process (csrss.exe), the local security authority subsystem (LSASS—\winnt\system32\lsass.exe), and the WinLogon process (\winnt\system32\winlogon.exe). The system account owns virtually every defined privilege (e.g., take ownership, reboot, create security token), and most files and Registry keys allow access to the system account. Even if those files and keys don't have access to Local System, a system account process can exercise the take-ownership privilege to gain access. Thus, the system account is more powerful than any local or domain account in terms of security potential on a local system.

A service's security context is an important consideration for service developers and systems administrators because that context has limitations. First, because the system account isn't associated with a particular user, its HKEY_CURRENT_USER key is that of the Default profile. In addition, a service running in the system account can't access the profile information for any other account. Second, even though a service has access to most local resources without needing to perform any special actions, the service has only limited access to network resources. Services in the system account can access only file and printer shares and named pipes 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 HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\LanmanServer\Parameters. In Win2K, the system account's security context is that of the machine account, so a service can also access network resources that explicitly grant access to the machine account of the computer on which the service is running.

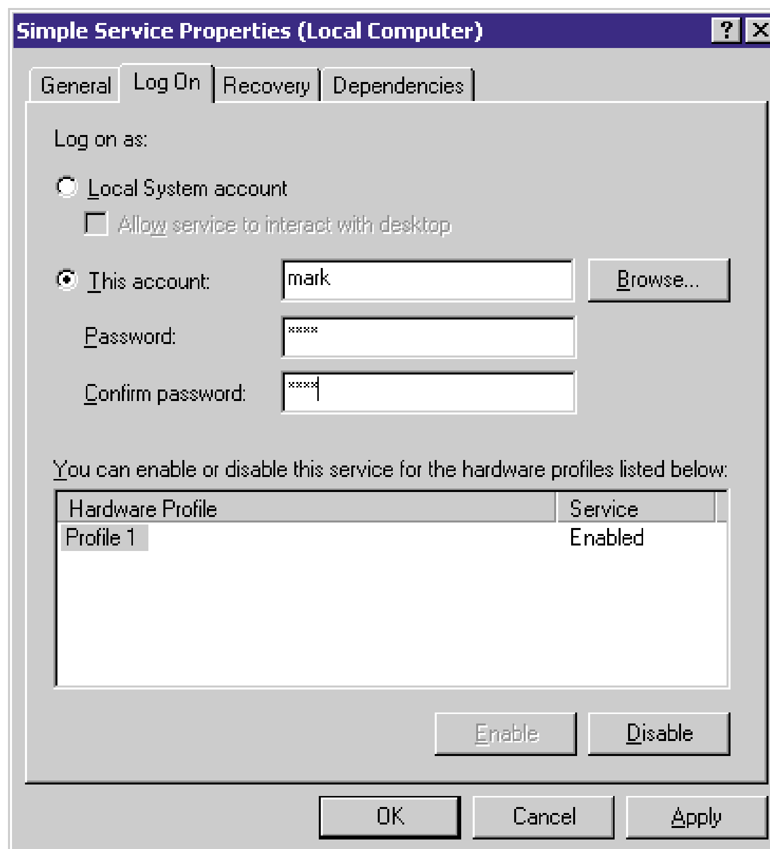
Another security-related services restriction isn't the direct result of being in the system account but rather is affected by the system account restriction. The Win32 subsystem associates every Win32 process with a window station. A window station contains the desktop interface, and only one window 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 all services run as part of the console session. Win32 names the visible window station WinSta0, and all interactive processes access WinSta0. Unless the system directs otherwise, the SCM associates services with Service-0x0-3e7$, a nonvisible window station that all noninteractive services share. The number 3e7 in this station's name is the logon session identifier that LSASS assigns to the logon session that the SCM uses for noninteractive services running in the system account. Services running with the default service window station therefore can't receive input from a user or display windows on the console. In fact, if a service were to present a dialog box on the window station, the service would appear to hang because a user wouldn't be able to see the dialog box or enter keyboard or mouse input to dismiss the box and let the service continue executing.

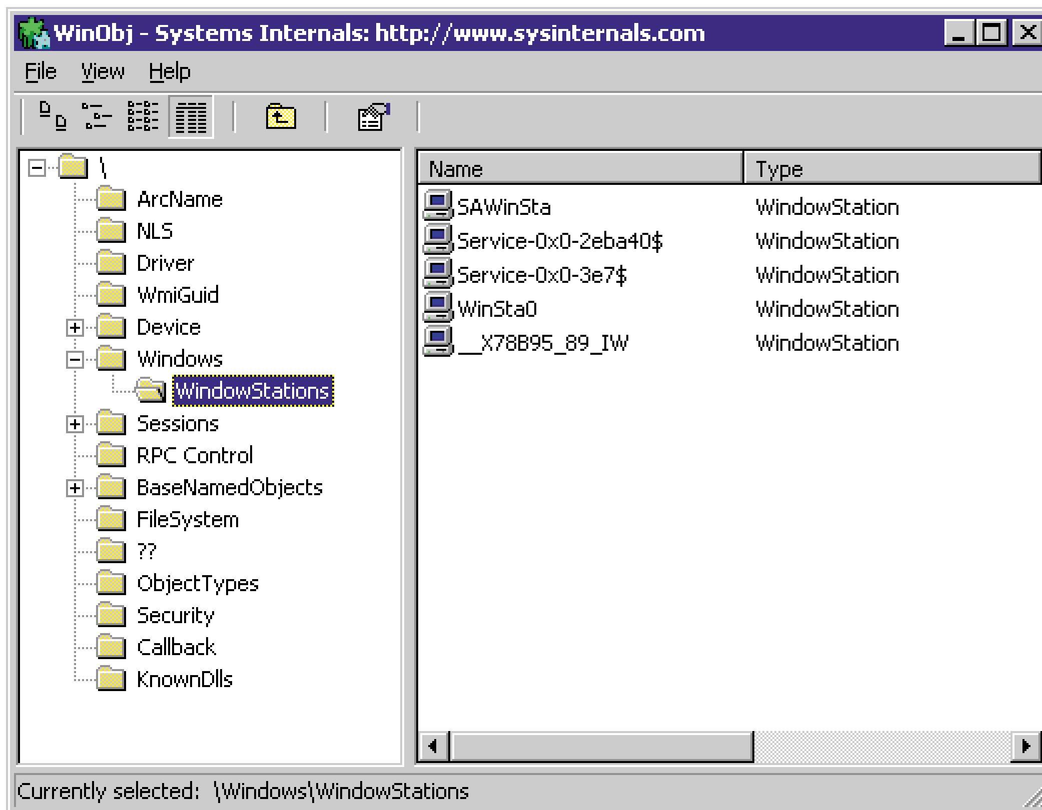
Most services experience no problems with any of the default security-related restrictions, but some services might need to interact with users through dialog boxes or access profile information for a particular user account. Therefore, the SCM provides two options to change the way services start. One option lets an installation application or SCP add the SERVICE_INTERACTIVE_PROCESS modifier to a service's Type parameter. When the SCM starts an interactive service, the SCM launches the service's process in the system account's security context but connects the service with WinSta0 rather than with the noninteractive service window station. This option lets the service display on the console dialog boxes and windows that can respond to user input.

The second option lets a service run in a user account rather than the system account. A service running in a user account has access to the account's HKEY_CURRENT_USER profile information and to any network resources that the user can access. When an SCP or installation program wants the service to run in a user account, the program specifies a user account and password, and the SCM starts the service process by logging the service on to the system as the specified user. Rather than run the process with the visible window station or the default service window station, the service runs in a window station whose name is the LSASS logon identifier that LSASS assigned for the service's logon session. As a result, the service doesn't have access to the console or user input; therefore, making a service run in a user account is incompatible with making the service interactive.

Noninteractive services also have a way to gain limited interaction with users. If a noninteractive service uses either the MB_DEFAULT_DESKTOP_ONLY or MB_SERVICE_NOTIFICATION flag, the service can display a message box. MB_SERVICE_NOTIFICATION displays the dialog box on the current active desktop, even if no user is logged on. MB_DEFAULT_DESKTOP_ONLY displays the dialog box on the interactive window station's default desktop.

Screen 2 displays the Win2K Services Microsoft Management Console (MMC) snap-in's dialog box, which lets you configure a service's logon parameters. Screen 3 shows the WinObj tool viewing the Object Manager directory, in which Win32 places window station objects. (You can download a free version of WinObj from the Systems Internals Web site at http://sysinternals.com/winobj.htm.) Visible in Screen 3 are the interactive window station (WinSta0), the noninteractive system service window station (Service-0x0-3e7$), and a noninteractive window station that the system assigned to a service process that has logged on as a user (Service-0x0-2eba40$).





The Service Control Manager

The SCM's executable file is \winnt\system32\services.exe in both Win2K and NT 4.0. Similarly to most service processes, this file runs as a Win32 console program. The Winlogon process starts the SCM early during the system boot. The SCM's startup function, SvcCtrlMain, launches services that are configured for automatic startup. SvcCtrlMain executes shortly after the screen switches to a blank desktop but generally before Winlogon loads the Graphical Identification and Authentication (GINA) interface that displays the logon dialog box. SvcCtrlMain first creates SvcCtrlEvent_A3752DX, a synchronization event that SvcCtrlMain initializes as nonsignaled. Only after the SCM completes the steps necessary to prepare SvcCtrlEvent_A3752DX to receive commands from SCPs does the SCM set the event to a signaled state. An SCP uses the OpenSCManager API, which ADVAPI32 supplies, to establish a dialog with the SCM. To prevent an SCP from trying to contact the SCM before the SCM initializes, OpenSCManager waits for SvcCtrlEvent_A3752DX to become signaled before establishing a dialog with the SCM.

Next, SvcCtrlMain calls ScCreateServiceDB, the function that builds the SCM's internal service database. ScCreateServiceDB reads and stores the contents of the MULTI_SZ value HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\ Control\ServiceGroupOrder\List, which lists the names and order of the defined service groups. A service's Registry key specifies an optional Group value if another service or device driver needs to control that service's startup ordering in relation to services from other groups. For example, the Win2K networking stack is built from the bottom up, so networking services must specify groups that order the stack later in the startup sequence than do groups to which networking device drivers belong. The SCM creates a group list that preserves the ordering of the groups that the SCM reads from the Registry. These groups include NDIS, TDI, Primary Disk, Keyboard Port, and Keyboard Class. Add-on and third-party applications

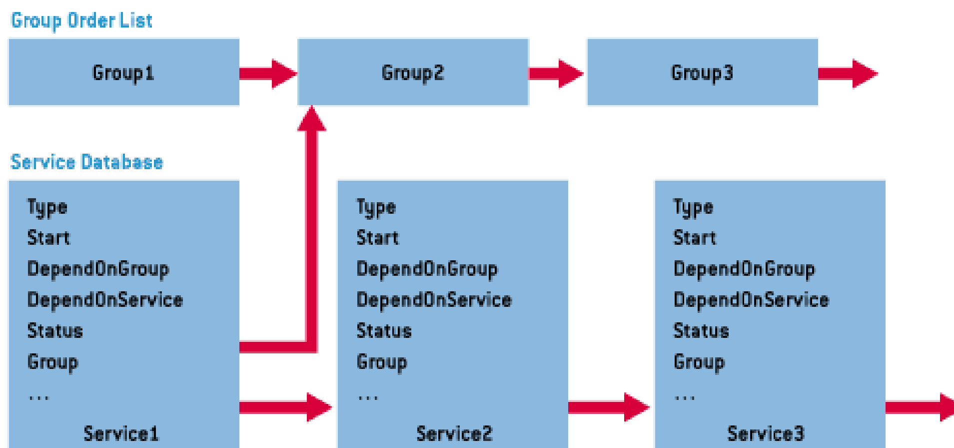
can define their own groups and add them to the SCM's list. Microsoft Transaction Server (MTS), for example, adds the MS Transactions group.

Next, ScCreateServiceDB scans the contents of:

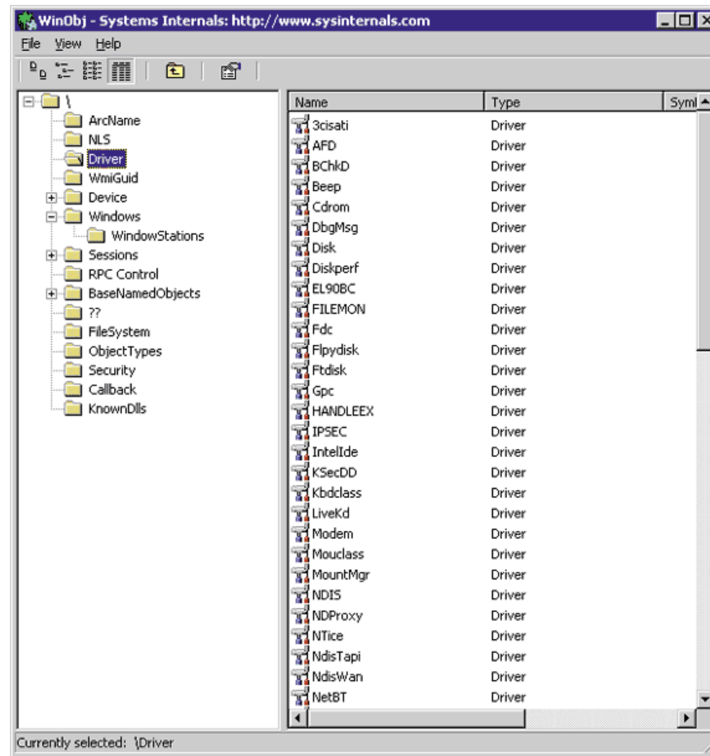
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services

and creates an entry in the SCM's service database for each key it encounters. Each entry includes all the service-related parameters that the system defines for a service, as well as fields that track the service's status. The SCM adds entries for device drivers as well as services because the SCM starts services and drivers marked as Auto Start and detects startup failures for drivers marked Boot and System Start. The kernel's I/O Manager loads drivers marked Boot and System Start before any user-mode processes execute; therefore, any drivers with these start types load before the SCM starts.

ScCreateServiceDB reads a service's Group value to determine the service's membership in a group, then associates the service with the group's entry in the group list that the SCM created earlier. The function also queries its DependOnGroup and DependOnService Registry values to read and record in the SCM's database a service's group and service dependencies. Figure 2 shows how the SCM organizes the service entry and group order lists. The service list appears in alphabetical order because the SCM creates the list from the Services Registry key, and Win2K stores Registry keys alphabetically.



During service startup, the SCM might need to call LSASS (e.g., 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 shows that LSASS has finished initializing. Winlogon also starts the LSASS process, so LSASS's initialization is concurrent with the SCM's initialization. Then, SvcCtrlMain calls ScGetBootAndSystemDriverState to scan the service database and look for Boot and System Start device-driver entries. ScGetBootAndSystemDriverState looks up a driver's name in the Object Manager's Driver namespace directory to determine whether a driver started successfully. When a device driver loads successfully, the I/O Manager inserts the driver's object in the namespace under the Driver directory; if a driver's name is not in this directory, then the driver hasn't loaded. Screen 4 shows WinObj viewing the contents of the Driver directory. If a driver doesn't load, the SCM looks for the driver's name in the list of drivers that the PnP_DeviceList API returns. PnP_DeviceList supplies the names of the drivers that the system's current hardware profile includes. SvcCtrlMain notes the names of drivers that haven't started and that are part of the current hardware profile in the ScFailedDrivers list.



Before starting the auto-start services, the SCM performs a few more steps. It creates its remote procedure call (RPC) named pipe (pipe\ntsvcs), then launches a thread to listen on the pipe for incoming messages from SCPs. Then, the SCM signals its initialization-complete event, SvcCtrlEvent_A3752DX. Registering a console application shutdown event handler with the Win32 subsystem process via RegisterServicesProcess prepares the SCM for system shutdown.

Back to Startup

Next month, I'll conclude this series with a description of how the SCM starts auto-start services, reacts to services that report errors during their initialization, and shuts down services. I'll describe how the Win32 subsystem shuts down the SCM, and how SCPs such as Win2K's Services MMC snap-in work. Then, I'll explore the SrvAny tool's internals. This utility lets you convert any executable file into a service. I'll also mention some roles the SCM fills that are unrelated to services.

I began this two-part series about Win32 services by presenting the structure of service applications, reviewing restrictions related to service user-account settings, and stepping through the first phases of the Service Control Manager's (SCM's) initialization. This time, I continue with a detailed description of how auto-start services initialize during the system boot. You'll also learn about the steps the SCM takes when a service fails during its startup, and about how the SCM shuts down services. Finally, I cover some new features of Windows 2000's services support, including service failure-recovery options.

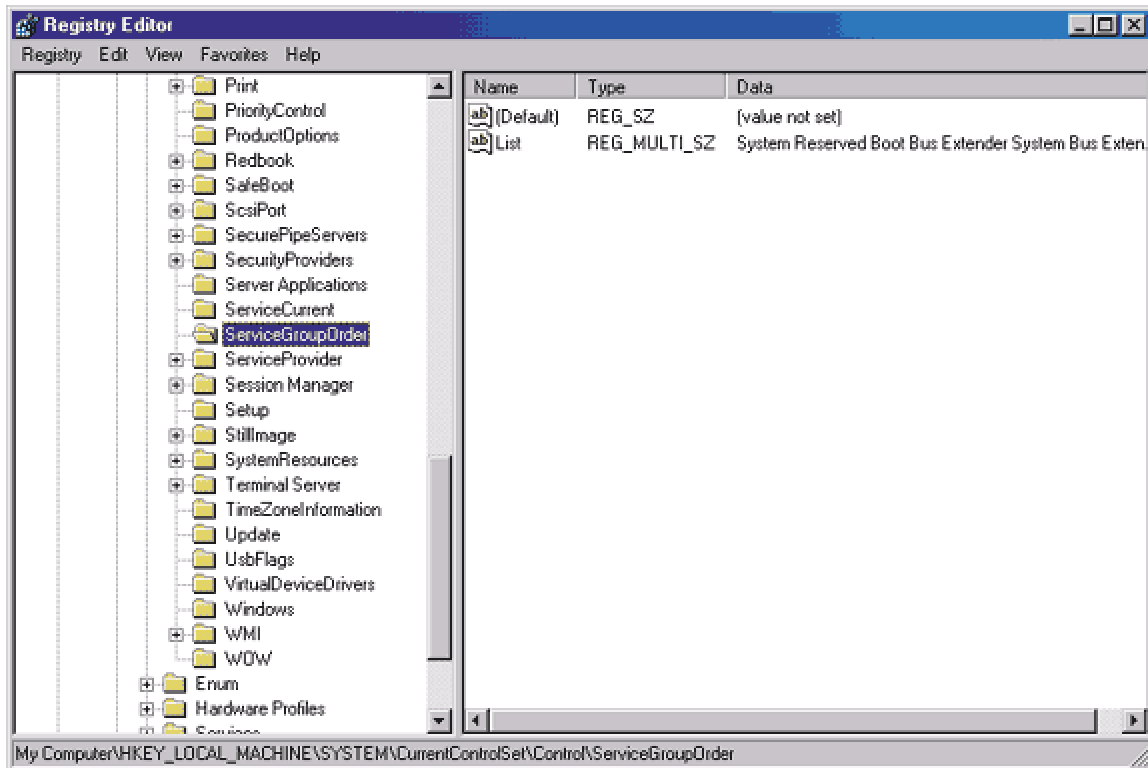
Service Startup

SvcCtrlMain (the SCM's startup function) invokes the SCM function ScAutoStartServices to start all services that have an auto-start Start value. (ScAutoStartServices also starts auto-start device drivers, but for the purposes of this article, when I use the term services, I mean services and drivers, unless I indicate otherwise.) ScAutoStartServices' algorithm for starting services in the correct order proceeds

in phases; each phase corresponds to a group, and phases proceed in the sequence that the group ordering stored in the:

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\ServiceGroupOrder

Registry value defines. As I explained in Part 1, a service belongs to a group if the service's Registry key has a Group value. The ServiceGroupOrder value, which Figure 1, page 58, shows, lists the names of groups in the order in which the SCM starts them. Thus, assigning a service to a group has no effect other than to fine-tune its startup with respect to other services that belong to different groups.



When a phase starts, ScAutoStartServices marks for startup all the service entries that belong to the phase's group. Then, ScAutoStartServices loops through the marked services to determine whether it can start each one. Part of the check ScAutoStartServices makes consists of determining whether a service has a dependency on another group; the DependOnGroup value in the service's Registry key specifies group dependencies. If a dependency exists, then the group on which the service is dependent must already have initialized, and at least one service of the group must have started successfully. If the service depends on a group that starts later in the group startup sequence than the service's group, the SCM notes a circular dependency error for the service and doesn't start the service. If the service depends on any services from its group that haven't yet started, then ScAutoStartServices skips over the service. If ScAutoStartServices is checking a Win32 service and not a device driver, ScAutoStartServices next tries to determine whether the service depends on one or more other services, and if so, whether those other services have already started. The DependOnService Registry value in a service's Registry key stores service dependencies.

Before ScAutoStartServices starts a service that has passed the function's dependencies check, ScAutoStartServices makes a final check to determine whether the service is part of the current boot

configuration. When a user boots the system in safe mode, the SCM ensures that either a name or a group identifies the service in the appropriate safe-boot Registry key. Two safe-boot keys, Minimal and Network, exist under HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\SafeBoot; which safe-boot key the SCM checks depends on which safe mode the user booted. If the user chose standard safe mode or safe mode with command prompt at the special boot menu (to access the special boot menu, you press F8 when prompted in the boot process), the SCM references the Minimal key. If the user chose networking-enabled safe mode, the SCM references the Network key. The existence of the Option string value under the SafeBoot key signals that the system booted in safe mode and lists the safe-mode type the user selected. (For more information about safe-mode booting options, see "Inside Win2K Reliability Enhancements, Part 1," August 1999.)

After the SCM decides to start a service, it calls ScStartService, which takes different steps to start services than to start device drivers. When ScStartService starts a Win32 service, the function first reads the ImagePath value from the service's Registry key to determine the name of the file that runs the service's process. Then, ScStartService examines the service's Type value; if this value is SERVICE_WIN32_SHARE_PROCESS, the SCM ensures that the process the service runs in, if already started, is logged in using the same account as specified for the service. A service's ObjectName Registry value stores the user account in which the service will run. A service with no ObjectName or with the LocalSystem ObjectName runs in the Local System account, an account with security privileges that I described in Part 1.

To verify that the service's process has not already started in a different account, the SCM checks to see whether the service's ImagePath value has an entry in the image database, an internal SCM database. If the image database doesn't have an entry for the service's ImagePath, the SCM creates the entry. Image database entries store a logon account name and an ImagePath value. When it creates a new image database entry, the SCM includes the logon account name for the service and the service's ImagePath value.

All services must have an ImagePath value; if not, the SCM doesn't start the service and generates an error stating that it couldn't find the service's path. If the SCM locates an existing image database entry with a matching ImagePath value, the SCM ensures that the user-account information for the service is the same as that stored in the database entry. Because a process can be logged on only as one account, the SCM reports an error if a service specifies an account name that is different from the account name of another service that has already started in the same process.

The SCM calls ScLogonAndStartImage to optionally log on a service and start the service's process. To log on services that don't run in the system account, the SCM calls the Local Security Authority Subsystem (LSASS—\winnt\system32\lsass.exe) function LsaLogonUser. LsaLogonUser requires a password, and the SCM signals to LSASS that the password for a service that doesn't run in the system account is stored as a services LSASS secret under the:

```
HKEY_LOCAL_MACHINE\SECURITY\Policy\Secrets
```

Registry key. When the SCM calls LsaLogonUser, the SCM specifies a service logon as the logon type, so LSASS looks up the password as a name in the form _SC_<service name> under the Secrets subkey. The SCM directs LSASS to store a logon password as a secret when a Service Control Program (SCP) configures a service's logon information. When the logon is successful, LsaLogonUser returns a handle to an access token to the SCM. Win2K uses access tokens to represent a user's security context, and the SCM later associates the token it received from LsaLogonUser with the process that implements the service.

After a successful logon, the SCM calls the UserEnv DLL's (\winnt\system32\userenv.dll) LoadUserProfile function to load the account's profile information, if the information isn't already loaded. LoadUserProfile loads the Registry hive (i.e., Registry file) that the user's profile key under:

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Winlogon\ProfileList

points to, making the hive the HKEY_CURRENT_USER key for the service.

An interactive service must open the WinSta0 window station, but before ScLogonAndStartImage lets an interactive service access WinSta0, the function confirms whether the:

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Windows\NoInteractiveServices

value is set. Administrators set this value to prevent interactive services from displaying windows on the console. This option is desirable in unattended server environments, where no user is present to respond to pop-ups from interactive services.

Next, ScLogonAndStartImage launches the service's process if the process hasn't already started (e.g., for another service). The SCM uses the CreateProcessAsUser Win32 API to start the process in a suspended state. The SCM next creates a named pipe through which it communicates with the service process and assigns the pipe the name \Pipe\Net\NetControlPipex, where x is a number that increments each time the SCM creates a pipe. The SCM uses the ResumeThread API to resume the service process and waits for the service to connect to its SCM pipe. If the Registry value:

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\ServicesPipeTimeout

exists, it determines the length of time that the SCM waits for a service to call StartServiceCtrlDispatcher and connect before the SCM gives up, terminates the process, and concludes that the service failed to start. If the ServicesPipeTimeout value 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 that the SCM created, 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 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 the SCM does when a service doesn't call StartServiceCtrlDispatcher within the timeout period). Instead, the SCM records an error in the System event log that the service failed to start in a timely manner.

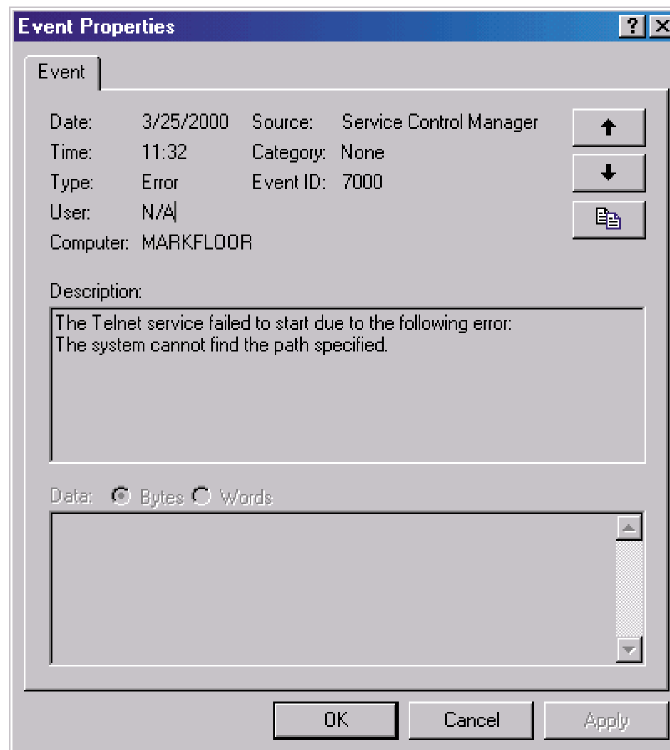
If the service the SCM starts with a call to ScStartService has a SERVICE_KERNEL_DRIVER or SERVICE_FILE_SYSTEM_DRIVER Value type, the service is actually a device driver. In that case, ScStartService calls ScLoadDeviceDriver to load the driver. ScLoadDeviceDriver enables the load driver security privilege for the SCM process and invokes the kernel service NtLoadDriver, passing the ImagePath value of the driver's Registry key to the function. Unlike services, drivers don't need to specify an ImagePath value. If a driver doesn't specify an ImagePath value, the SCM concatenates the driver's name with \winnt\system32\drivers to build an ImagePath.

ScAutoStartServices continues looping through the services in a group until all the group's services have either started or generated dependency errors. This looping process is how the SCM automatically orders services within a group according to their DependOnService dependencies. The SCM starts services that other services depend on in earlier loops, leaving the dependent services for subsequent loops. The SCM ignores Tag values for Win32 services. You might come across Tag values in Registry keys under HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services. The I/O Manager uses Tag values to order device-driver startup for boot and system-start drivers.

After the SCM completes the startup phases for all the groups that the ServiceGroupOrder value lists, the SCM performs a startup phase for services that belong to groups that the value doesn't list. Finally, the SCM completes a final startup phase for services that don't belong to any group.

Startup Errors

If a driver or 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 isn't specified, the SCM simply ignores the error and continues processing service startups. If the ErrorControl value is SERVICE_ERROR_NORMAL (1), then the SCM writes an event to the System event log stating that the service failed to start and specifies the reason. The SCM includes in the event log the textual representation of the Win32 error code that the service returns to the SCM as the reason for the startup failure. Figure 2 shows the Event Viewer utility displaying an example event-log entry that reports a service startup error.



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, then calls the internal function ScRevertToLastKnownGood. ScRevertToLastKnownGood switches the system's Registry configuration to the Last Known Good version, which is the boot configuration with which the system last booted successfully. Then, the SCM restarts the system using the

NtShutdownSystem system service, which implements in the kernel. If the Registry configuration was already set to Last Known Good, then the system simply reboots.

Accepting the Boot and Last Known Good

In addition to making the SCM responsible for starting services, the system charges the SCM with determining when the system's Registry configuration:

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet

should be saved as the LastKnownGood control set. CurrentControlSet contains the Services key as a subkey, so the CurrentControlSet key includes the Registry representation of the SCM database. CurrentControlSet also contains the Control key, which stores many kernel- and user-mode subsystem configuration settings. By default, a successful boot consists of auto-start services successfully starting and a successful user logon. A boot fails if 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 successfully started auto-start services, but Winlogon (\winnt\system32\winlogon.exe) must notify the SCM when a user has successfully logged on. Winlogon invokes the ADVAPI32 (\winnt\system32\advapi32.dll) function NotifyBootConfigStatus when a user logs on, and NotifyBootConfigStatus sends a message to the SCM. Following the successful startup of auto-start services and receipt of the NotifyBootConfigStatus message (whichever comes last), 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 can accept and process transactions. A developer imposes its successful-boot definition by writing a boot verification program and installs the program by pointing to the program's on-disk location with the:

HKEY_LOCAL_MACHINESYSTEM\CurrentControlSet\Control\Boot VerificationProgram

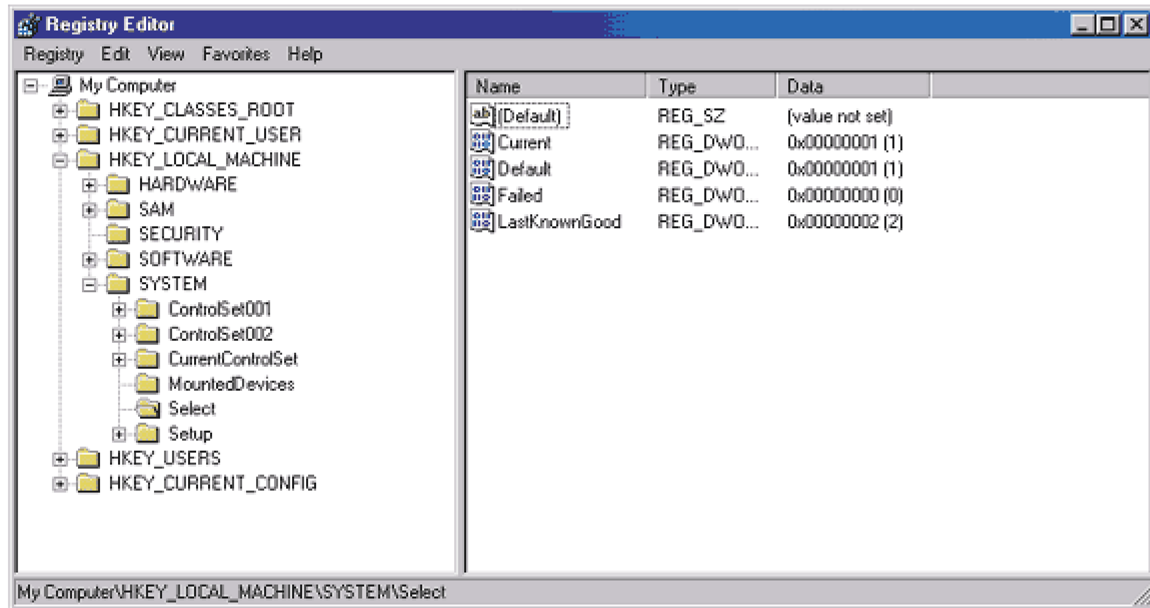
Registry value. In addition, a proprietary boot verification program's installation must set:

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\WindowsNT\CurrentVersion\Winlogon\ReportBootOk

to 0 to disable Win logon's call to NotifyBootConfigStatus. When a proprietary boot verification program exists, the SCM launches the program after starting auto-start services, then waits for the program's call to NotifyBootConfigStatus before saving the LastKnownGood control set.

Win2K maintains several copies of CurrentControlSet, and the CurrentControl Set key is actually a symbolic Registry link that points to one of the copies. Control sets have names in the form HKEY_LOCAL_MACHINE\SYSTEM\ControlSet nnn, where nnn is a number such as 001 or 002. The HKEY_LOCAL_MACHINESYSTEM\Select key contains values that identify the role of each control set. For example, if CurrentControlSet points to ControlSet001, then Current under Select has a value of 1. Last Known Good contains the number of the Last KnownGood control set, which is the control set the system last used to boot successfully. You might have the Failed control set on your system. Failed points to the last control set for which the boot was unsuccessful and indicates that the system

aborted in favor of attempting to boot with the LastKnownGood control set. Figure 3 displays a system's control sets and Select values.



NtInitializeRegistry synchronizes the contents of the LastKnownGood control set with CurrentControlSet's tree. After a system's first successful boot, LastKnownGood doesn't exist, so the system creates a LastKnownGood control set. If the LastKnownGood tree already exists, the system simply updates the tree by synchronizing it with CurrentControlSet.

The Win2K SCM's reliance on NtInitializeRegistry to update Last Known Good differs from the NT 4.0 SCM's behavior. In NT 4.0, the SCM is fully responsible for managing control sets. Before the NT 4.0 SCM starts auto-start services, it copies the CurrentControlSet to a new key, Clone. After the boot succeeds, the SCM makes a copy of the Clone key and labels the key LastKnownGood. Win2K thereby optimizes performance because whereas NT 4.0 copies CurrentControlSet twice, Win2K usually performs only an update, not a copy operation.

Last Known Good is helpful in situations in which a change to CurrentControlSet, such as the modification of a system performance-tuning value under:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control
```

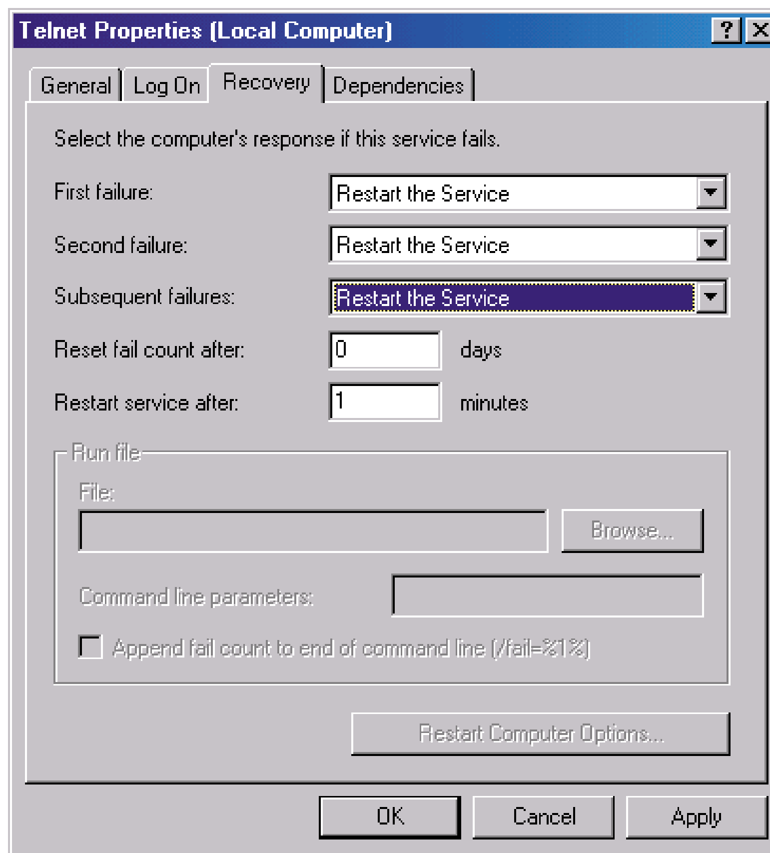
or the addition of a service or device driver, causes the subsequent boot to fail. Users can press F8 early in the boot process to bring up a menu that lets them direct the boot to use the LastKnownGood control set, which rolls the system's Registry configuration back to the way it was the last time the system booted successfully.

Service Failures

In NT 4.0, a service that fails after it starts successfully does so silently. An administrator has no way of knowing, without checking manually or using third-party service-monitoring utilities, that the service's process has exited. Win2K introduces service failure-action capability, a feature that the SCM implements. A Win2K service can have optional FailureAction and FailureCommand values in its Registry key; the SCM records these values 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 that the services' failure-related Registry values specify.

Actions that a service can configure for the SCM include restarting the service, running a program, or rebooting the computer. In addition, a service can specify what failure actions take place the first, second, and subsequent times the service process fails and can indicate a period during which the SCM must wait before restarting the service if the service asks to be restarted. For example, the IIS Administrator service's failure action results in the SCM running the IISReset application, which performs cleanup work and restarts the service. You can easily manage a service's recovery actions from the Recovery tab of the service's Properties dialog box in the Services Microsoft Management Console (MMC) snap-in, as Figure 4, page 62, shows.



Service Shutdown

When Winlogon calls the Win32 ExitWindowsEx API, ExitWindowsEx sends a message to CSRSS, the Win32 subsystem process, that invokes CSRSS's shutdown routine. CSRSS loops through all active processes and notifies them that the system is shutting down. For every system process except the SCM, CSRSS waits for the number of seconds that HKEY_USERS\.DEFAULT\Control Panel\Desktop\WaitToKillAppTimeout specifies (the default is 20 seconds) for the process to exit before moving to the next process. When CSRSS encounters the SCM process, CSRSS notifies the SCM that the system is shutting down and employs a timeout specific to the SCM. The process ID that the SCM used when it registered with CSRSS during system initialization lets CSRSS recognize the SCM. The SCM's timeout differs from that of other processes because the SCM communicates

with services that need to perform cleanup when they shut down; thus, an administrator might need to tune only the SCM's timeout. The SCM's timeout value resides in the:

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\WaitToKillServicesTimeout

Registry value and defaults to 30 seconds.

The SCM's shutdown handler sends shutdown notifications to all services that request shutdown notification when they initialize with the SCM. The SCM function ScShutdownAllServices loops through the SCM services database searching for services that request shutdown notification and sends each such service a shutdown command. For each service to which ScShutdownAllServices 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 records. After ScShutdownAllServices sends shutdown messages, the SCM waits until one of the services notified of shutdown exits or until the largest wait hint's timeout passes.

If the wait hint expires before a service exits, the SCM determines whether one or more of the services upon which it was waiting has sent a message notifying the SCM that the service is progressing in its shutdown process. If at least one service made progress toward shutdown, the SCM waits again for the duration of the largest wait hint's timeout. The SCM continues this wait loop until either all the services have exited or none of the services upon which it is waiting has sent notification of shutdown progress within the wait hint's timeout.

After the SCM has directed services to shut down and is waiting for the services to exit, CSRSS waits for the SCM to exit. If CSRSS's wait ends before the SCM exits (i.e., the WaitToKillServicesTimeout expires), CSRSS simply continues the shutdown process. Thus, CSRSS leaves running those services (as well as the SCM) that fail to shut down in a timely manner as the system shuts down. Unfortunately, administrators have no way of knowing whether they should raise the WaitToKillServicesTimeout value on systems on which services are not getting a chance to completely shut down before the system shuts down.

Shared Service Processes

Running every service in its own process, rather than having services share a process when sharing is possible, wastes system resources. However, when services share a process, any service in the shared process that has a bug that causes the process to exit also causes all the services in the process to terminate. Win2K includes many built-in services, so Microsoft chose a mix of approaches to maximize system stability and minimize resource usage.

In both Win2K and NT 4.0, the SCM process hosts many services, including the EventLog service, the file-server service (LanmanServer), and the LAN Manager name-resolution service. Table 1 lists the services that the SCM hosts in Win2K (not every service is active on every system). You can use the Tlist program from the Win2K support tools CD-ROM with the /s option to obtain a list of which services run in processes.

TABLE 1: Windows 2000 Services That Run in the SCM

Service	Service Description
Alerter	Notifies selected users and computers of administrative alerts.
AppMgmt	Provides software installation services such as Assign, Publish, and

	Remove.
Browser	Maintains an up-to-date list of computers on your network and supplies the list to programs that request it.
Dhcp	Manages network configuration by registering and updating IP addresses and DNS names.
DmServer	Enables the Logical Disk Manager (LDM) Watchdog Service.
DnsCache	Resolves and caches DNS names.
EventLog	Logs event messages that programs and Windows issue. EventLog reports contain information that can be useful in diagnosing problems. Event Viewer displays reports.
LanmanServer	Provides Remote Procedure Call (RPC) support and file, print, and named pipe sharing.
LanmanWorkstation	Provides network connections and communications.
LmHosts	Enables support for NetBIOS over TCP/IP (NetBT) service and NetBIOS name resolution.
Messenger	Sends and receives messages that administrators or the Alerter service transmit.
PlugPlay	Manages device installation and configuration and notifies programs of device changes.
ProtectedStorage	Provides protected storage for sensitive data (e.g., private keys) to prevent access by unauthorized services, processes, or users.
SecLogon	Enables starting processes under alternative credentials.
TrkSvr	Stores information to track files moved between volumes for each volume in the domain.
TrkWks	Sends notifications of files moving between NTFS volumes in a network domain.
W32Time	Sets the computer clock.
Wmi	Provides systems management information to and from drivers.

In NT 4.0, the SCM is the only process that the system uses to host multiple built-in services. Several other services, such as the Remote Procedure Call Subsystem service (RpcSs), the Telephony API service (TapiSrv), and Remote Access Manager (RasMan), use separate processes.

Because of the increased number of built-in services in Win2K, Microsoft decided to include more services in the Win2K SCM and to add new processes to act as service hosts. In NT 4.0, the LSASS process included only the Netlogon service. In Win2K, several security-related services (e.g., the Security Accounts Manager Subsystem—SamSs—the Netlogon service, and the Win2K Policy Agent service) share the LSASS process.

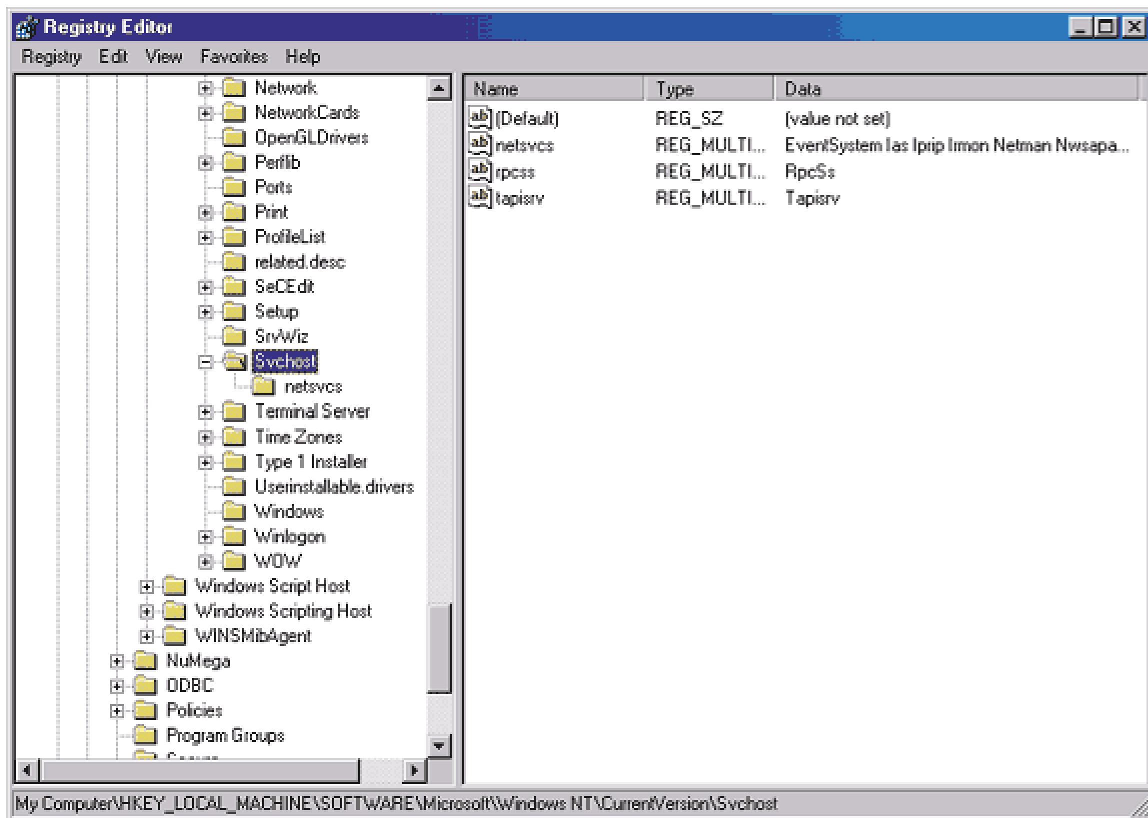
Win2K also introduces Service Host (SvcHost—\winnt\system32\svchost.exe), an application that exists solely to execute in processes that host services. A Win2K system can start multiple instances of SvcHost running in different processes. Services that run in SvcHost processes include TapiSrv,

RpcSs, and RasMan. Win2K implements services that run in SvcHost as DLLs and includes an ImagePath definition of the form \%systemroot%\system32\svchost.exe -k netsvcs in the services' Registry key. The services' Registry key must also have a Registry value of ServiceDll under a Parameters subkey that points to the services' DLL file.

All services that share a common SvcHost process specify the same parameter (-k netsvcs in the previous example) so that they each have the same entries in the SCM's image database. When the SCM first encounters a service during service startup with a SvcHost ImagePath and a particular parameter, the SCM creates a new image database entry and launches a SvcHost process with the parameter. The new SvcHost process looks under:

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\SvcHost

for a value that has the same name as the service's parameter. SvcHost interprets the contents of the value as a list of service names; when it registers with the SCM, SvcHost notifies the SCM that the SvcHost process is hosting those services. Figure 5 presents an example of a SvcHost Registry key that shows that a SvcHost process started with the -k netsvcs parameter is prepared to host many different network-related services.



When the SCM encounters a SvcHost service during service startup with an ImagePath that matches an entry that already appears in the SCM's image database, the SCM doesn't launch a second process. Rather, the SCM sends a start command for the service to the SvcHost process that is 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 the service's process to start the service.

Service Control Programs

SCPs are standard Win32 applications that use SCM APIs that ADVAPI32 exports. APIs that the SCM implements include:

- CreateService
- OpenService
- StartService
- ControlService
- QueryServiceStatus
- DeleteService

To use a SCM API, an SCP must first call the OpenSCManager API to open a communications channel to the SCM. At the time of the open call, the SCP must specify the types of actions it wants to perform. For example, if an SCP wants to enumerate and display the services in the SCM's database, the SCP requests enumerate-service access in its call to OpenSCManager. As the SCM initializes, it creates an internal object that represents the SCM database. The SCM uses the Win2K security APIs to protect the internal object, a security descriptor that specifies which accounts can open the object with which access permissions. For example, the security descriptor specifies that the Everyone group (of which every account is a member) has permission to open the SCM internal object with enumerate-service access. However, only administrators have permission to open the object with the access required to create or delete a service.

The SCM implements security for services, as it does for the SCM database. When an SCP uses the CreateService API to create a service, the SCP specifies a security descriptor that the SCM internally associates 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 reads the value when it scans the Registry's Services key during initialization. This setup ensures that the security settings persist across reboots. Just as an SCP must specify in its call to OpenSCManager what types of access the SCP wants to the SCM database, an SCP must tell the SCM in a call to OpenService what access the SCP wants to a service. 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 that you're likely most familiar with is the Control Panel Services applet in NT 4.0 and the Services MMC snap-in in Win2K. The NT 4.0 Control Panel Services applet implements its SCP in the \winnt\system32\srvmgr.cpl library, and in Win2K, the SCP resides in \winnt\system32\filemgr.dll. The Microsoft Windows NT Server 4.0 Resource Kit and the Microsoft Windows 2000 Resource Kit include sc.exe, a command-line SCP.

SCPs sometimes layer service policy over services that the SCM implements. A good example is the timeout that the Services MMC snap-in implements when you start a service manually. The snap-in displays a progress bar that represents the service's startup progress. 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 adjusting their configuration status to reflect their progress as they respond to SCM commands such as the start command. SCPs query services' status with the QueryServiceStatus API. Thus, SCPs can tell when a service actively updates its status or appears to be hung, and the SCM can take action to notify users about the service's activity.

SrvAny

The Win2K and NT 4.0 resource kits include the SrvAny utility, which lets you run any application as a service. (For information about how to use SrvAny, see Mark Minasi, This Old Resource Kit, February 2000 and March 2000.) SrvAny is similar to SvcHost—both are generic service-host applications. As in SvcHost, a SrvAny process reads the path of the service file that it loads from the Parameters subkey of the service's Registry key. When SrvAny starts, it notifies the SCM that it is hosting a particular service. Then, when SrvAny receives a start command, it launches the service executable file as a child process. Because the child process receives a copy of the SrvAny process' access token and a reference to the same window station, the executable file runs in the same security account and with the same interactivity setting that you specified when you configured the SrvAny process. Unlike SvcHost, however, SrvAny services don't have the share-process Type value. Therefore, each application you install as a service in SrvAny runs in a separate process with a different instance of the SrvAny host program.

Network Drive Letters

In addition to its role as a services interface, the SCM has another responsibility: It notifies GUI applications in a system whenever the system creates or deletes a network drive-letter connection. The SCM waits for the LAN Manager Workstation service to signal the ScNetDrvMsg named event. (The Workstation service signals ScNetDrvMsg whenever an application assigns a drive letter to a remote network share or deletes a remote-share drive letter assignment.) When the Workstation service signals the event, the SCM uses the GetDriveType Win32 API to query the list of connected network drive letters. If the list changes across the event signal, the SCM sends a type WM_DEVICECHANGE Windows broadcast message. The SCM uses either DBT_DEVICEREMOVECOMPLETE or DBT_DEVICEARRIVAL as the message's subtype. This message is intended primarily for Windows Explorer, so that it can update any open My Computer windows to show the presence or absence of a network drive letter.

A Better Handle on Services

Although the SCM remained mostly unchanged from NT 4.0 to Win2K, it picked up the powerful new capability of detecting service process failures and taking recovery steps to restart services or run arbitrary programs. In addition, Win2K's use of the SvcHost application reduces the overhead otherwise incurred with the increase in the number of built-in Win2K services. This information about service startup and shutdown can help you better understand what is going on behind the scenes so that you can troubleshoot service-related problems you might encounter.