lpServiceName is the name used for future references to the service and is one of the logical service names specified in the dispatch table in the StartServiceCtrlDispatcher call. Notice that there is a separate CreateService call for each logical service.

lpDisplayName is the name displayed to the user to represent the service in the Services administrative tool (accessed from the Control Panel under Administrative Tools) and elsewhere. You will see this name entered immediately after a successful CreateService call.

dwDesiredAccess can be SERVICE_ALL_ACCESS or combinations of GENERIC_READ, GENERIC_WRITE, and GENERIC_EXECUTE. See the MSDN documentation for additional details.

dwServiceType has values as in Table 13–1.

dwStartType specifies how the service is started. SERVICE_DEMAND_START is used in our examples, but other values (SERVICE_BOOT_START and SERVICE_SYSTEM_START) allow device driver services to be started at boot time or at system start time, and SERVICE_AUTO_START specifies that a service is to be started at machine start-up.

lpBinaryPathName gives the service's executable as a full path; the .exe extension is necessary. Use quotes if the path contains spaces.

Other parameters specify account name and password, groups for combining services, and dependencies when there are several interdependent services.

Service configuration parameters of an existing service can be changed with ChangeServiceConfig and ChangeServiceConfig2, which is simpler and is not, perhaps for that reason, called ChangeServiceConfigEx. Identify the service by its handle, and you can specify new values for most of the parameters. For example, you can provide a new dwServiceType or dwStartType value but not a new value for dwAccess.

There is also an OpenService function to obtain a handle to a named service. Use DeleteService to unregister a service from the SCM and CloseServiceHandle to close SC_HANDLEs.

Starting a Service

A service, once created, is not running. Start the *ServiceMain()* function by specifying the handle obtained from CreateService along with the argc, argv command line parameters expected by the service's main function (that is, the function specified in the dispatch table).

```
BOOL StartService (
   SC_HANDLE hService,
   DWORD argc,
   LPTSTR argv[])
```

Controlling a Service

Control a service by telling the SCM to invoke the service's control handler with the specified control.

```
BOOL ControlService (
   SC_HANDLE hService,
   DWORD dwControlCode,
   LPSERVICE_STATUS lpServStat)
```

The interesting dwControlCode values for our examples are:

SERVICE_CONTROL_STOP

SERVICE_CONTROL_PAUSE

SERVICE_CONTROL_CONTINUE

SERVICE_CONTROL_INTERROGATE

SERVICE_CONTROL_SHUTDOWN

or a user-specified value in the range 128—255. Additional named values notify a service that start-up values have changed or there are changes related to binding.

SERVICE_CONTROL_INTERROGATE tells the service to report its status with SetServiceStatus, but it's of limited use, as the SCM receives periodic updates.

lpServStat points to a SERVICE_STATUS structure that receives the current status. This is the same structure as that used by the SetServiceStatus function.

Querying Service Status

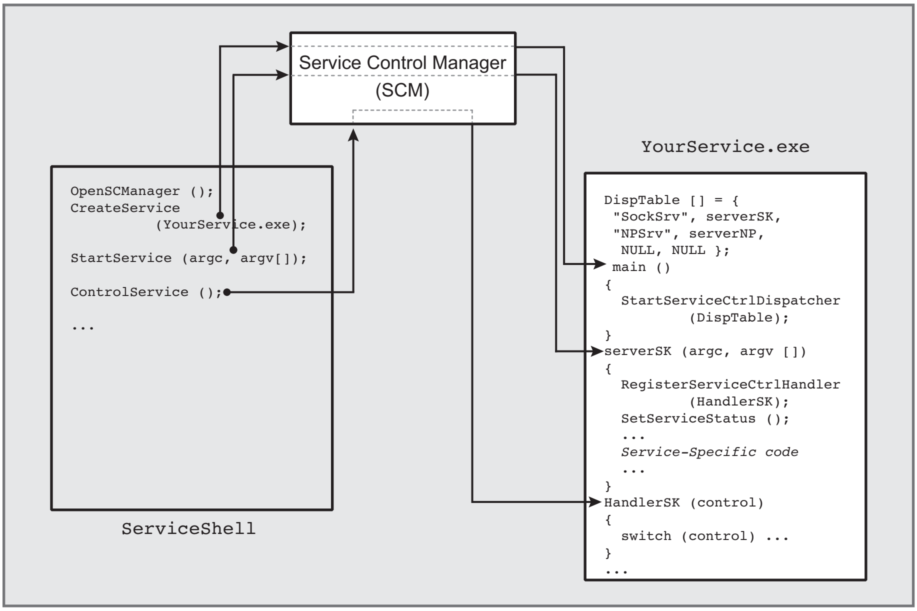
Obtain a service's current status in a SERVICE_STATUS structure with the following:

```
BOOL QueryServiceStatus (
   SC_HANDLE hService,
   LPSERVICE_STATUS lpServiceStatus)
```

There's a distinction between calling QueryServiceStatus, which gets the current status information from the SCM, and ControlService with a SERVICE_CONTROL_INTERROGATE control code. The former tells the service to update the SCM rather than the application program.

Summary: Service Operation and Management

Figure 13–1 shows the SCM and its relation to the services and to a service control program, such as the one in Program 13–3 in the next section. In particular, a service must register with the SCM, and all commands to the service pass through the SCM.



```
OpenSCManager ();
CreateService
          (YourService.exe);

StartService (argc, argv[]);

ControlService ();

...
```

```
DispTable [] = {
 "SockSrv", serverSK,
 "NPSrv", serverNP,
  NULL, NULL };
 main ()
{
  StartServiceCtrlDispatcher
          (DispTable);
}
serverSK (argc, argv [])
{
  RegisterServiceCtrlHandler
          (HandlerSK);
  SetServiceStatus ();
  ...
  Service-Specific code
  ...
}
HandlerSK (control)
{
  switch (control) ...
}
...
```

Figure 13–1 Controlling Windows Services through the SCM

Example: A Service Control Shell

You can control Windows Services from the Administrative Tools, where there is a Services icon. Alternatively, you can control services from the Windows command `sc.exe`. Finally, you can control a service from within an application, as illustrated in the next example, `ServiceShell` (Program 13–3), which is a modification of Chapter 6's `JobShell` (Program 6–3).

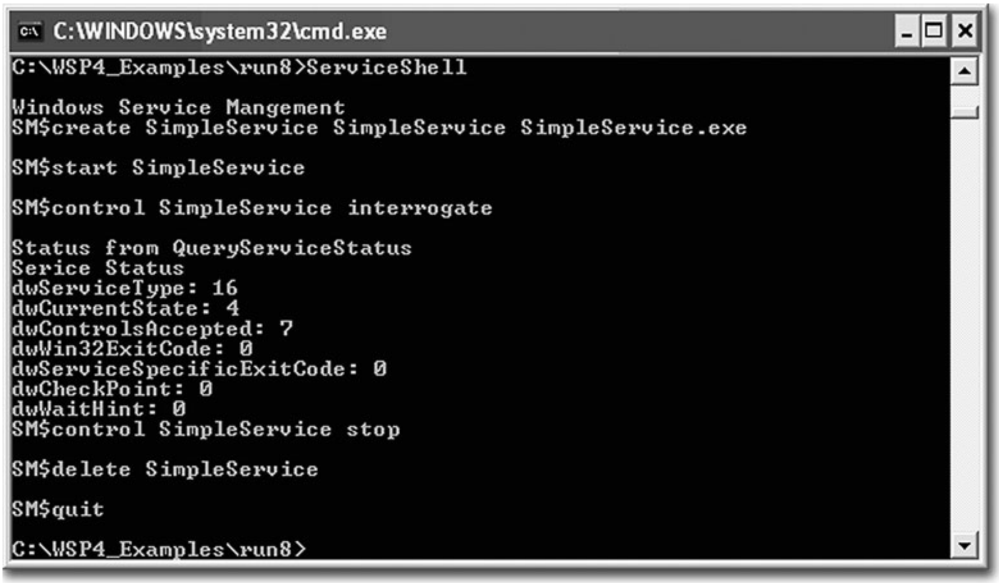
This example is intended to show how to control services from a program; it does not supplant `sc.exe` or the Services Administrative tool.

Program 13–3 `ServiceShell`: A Service Control Program

```
/* Chapter 13 */
/* ServiceShell.c  Windows Service Management shell program.
   This program modifies Chapter 6's Job Management program,
   managing services rather than jobs. */
/* Illustrates service control from a program
   In general, use the sc.exe command or the "Services"
   Administrative tools */
/* commandList supported are:
      create    Create a service
      delete    Delete a service
      start  Start a service
      control   Control a service */
#include "Everything.h"

static int Create   (int, LPTSTR *, LPTSTR);
static int Delete   (int, LPTSTR *, LPTSTR);
static int Start    (int, LPTSTR *, LPTSTR);
static int Control  (int, LPTSTR *, LPTSTR);

static SC_HANDLE hScm;
static BOOL debug;

int _tmain (int argc, LPTSTR argv[])
{
   BOOL exitFlag = FALSE;
   TCHAR command[MAX_COMMAND_LINE+10], *pc;
   DWORD i, locArgc; /* Local argc */
   TCHAR argstr[MAX_ARG][MAX_COMMAND_LINE];
   LPTSTR pArgs[MAX_ARG];

   debug = (argc > 1); /* simple debug flag */
   /*  Prepare the local "argv" array as pointers to strings */
   for (i = 0; i < MAX_ARG; i++) pArgs[i] = argstr[i];
   /*  Open the SC Control Manager on the local machine,
      with the default database, and all access. */
   hScm = OpenSCManager (NULL, SERVICES_ACTIVE_DATABASE,
         SC_MANAGER_ALL_ACCESS);

   /*  Main command processing loop  */
   _tprintf (_T("\nWindows Service Management"));
   while (!exitFlag) {
      _tprintf (_T("\nSM$"));
      _fgetts (command, MAX_COMMAND_LINE, stdin);
      /*  Replace the new line character with a string end. */
      pc = _tcschr (command, _T('\n')); *pc = _T('\0');

      if (debug) _tprintf (_T("%s\n"), command);
      /*  Convert the command to "argc, argv" form. */
      GetArgs (command, &locArgc, pArgs);
      CharLower (argstr[0]);  /* The command is case-insensitive */

      if (debug) _tprintf (_T("\n%s %s %s %s"), argstr[0], argstr[1],
         argstr[2], argstr[3]);

      if (_tcscmp (argstr[0], _T("create")) == 0) {
         Create (locArgc, pArgs, command);
      }
      else if (_tcscmp (argstr[0], _T("delete")) == 0) {
         Delete (locArgc, pArgs, command);
      }
      else if (_tcscmp (argstr[0], _T("start")) == 0) {
         Start (locArgc, pArgs, command);
      }
      else if (_tcscmp (argstr[0], _T("control")) == 0) {
         Control (locArgc, pArgs, command);
      }
      else if (_tcscmp (argstr[0], _T("quit")) == 0) {
         exitFlag = TRUE;
      }
      else _tprintf (_T("\nCommand not recognized"));
   }

   CloseServiceHandle (hScm);
   return 0;
}

int Create (int argc, LPTSTR argv[], LPTSTR command)
{
   /*  Create a new service as a "demand start" service:
      argv[1]: Service Name
      argv[2]: Display Name
      argv[3]: binary executable */
   SC_HANDLE hSc;
   TCHAR executable[MAX_PATH+1],
      quotedExecutable[MAX_PATH+3] = _T("\"");

   /* You need full path name, add quotes if there are spaces */
   GetFullPathName (argv[3], MAX_PATH+1, executable, NULL);
   _tcscat(quotedExecutable, executable);
   _tcscat(quotedExecutable, _T("\""));
   if (debug) _tprintf (_T("\nService Full Path Name: %s"),
         executable);

   hSc = CreateService (hScm, argv[1], argv[2],
      SERVICE_ALL_ACCESS, SERVICE_WIN32_OWN_PROCESS,
      SERVICE_DEMAND_START, SERVICE_ERROR_NORMAL,
      quotedExecutable, NULL, NULL, NULL, NULL, NULL);
      CloseServiceHandle (hSc); /* No need to retain the handle as
                             OpenService will query the service DB */
   return 0;
}

/*  Delete a service
      argv[1]: ServiceName to delete  */
int Delete (int argc, LPTSTR argv[], LPTSTR command)
{
   SC_HANDLE hSc;

   if (debug) _tprintf (_T("\nAbout to delete service: %s"), argv[1]);
   hSc = OpenService(hScm,  argv[1], DELETE);
   DeleteService (hSc);
   CloseServiceHandle (hSc);
   return 0;
}

/*  Start a named service.
      argv[1]: Service name to start */
int Start (int argc, LPTSTR argv[], LPTSTR command)
{
   SC_HANDLE hSc;
   TCHAR workingDir[MAX_PATH+1];
   LPTSTR argvStart[] = {argv[1], workingDir};

   GetCurrentDirectory (MAX_PATH+1, workingDir);

   /* Get a handle to service named on the command line (argv[1]) */
   hSc = OpenService(hScm,  argv[1], SERVICE_ALL_ACCESS);

   /*  Start the service with one arg, the working directory */
   /*  The service name is from the program command line (argv[1]) */
    */
   StartService (hSc, 2, argvStart);
   CloseServiceHandle (hSc);

   return 0;
}

/*  Control a named service.
      argv[1]:  Service name to control
      argv[2]:  Control command (case insenstive):
                stop
                pause
                resume
                interrogate
                user  user defined
                */
static LPCTSTR commandList[] =
   {  _T("stop"), _T("pause"), _T("resume"),
      _T("interrogate"), _T("user") };
static DWORD controlsAccepted[] = {
   SERVICE_CONTROL_STOP, SERVICE_CONTROL_PAUSE,
   SERVICE_CONTROL_CONTINUE, SERVICE_CONTROL_INTERROGATE, 128 };

int Control (int argc, LPTSTR argv[], LPTSTR command)
{
   SC_HANDLE hSc;
   SERVICE_STATUS sStatus;
   DWORD dwControl, i;
   BOOL found = FALSE;

   if (debug) _tprintf (_T("\nAControl service: %s"), argv[1]);

   for (i= 0;
         i < sizeof(controlsAccepted)/sizeof(DWORD) && !found; i++)
      found = (_tcscmp (commandList[i], argv[2]) == 0);
   if (!found) {
      _tprintf (_T("\nIllegal Control command %s"), argv[1]);
      return 1;
   }
   dwControl = controlsAccepted[i-1];
   if (dwControl == 128) dwControl = _ttoi (argv[3]);
   if (debug) _tprintf (_T("\ndwControl = %d"), dwControl);

   hSc = OpenService(hScm,  argv[1],
      SERVICE_INTERROGATE | SERVICE_PAUSE_CONTINUE |
      SERVICE_STOP | SERVICE_USER_DEFINED_CONTROL |
      SERVICE_QUERY_STATUS );

   ControlService (hSc, dwControl, &sStatus);

   if (dwControl == SERVICE_CONTROL_INTERROGATE) {
      QueryServiceStatus (hSc, &sStatus);
```

```
        _tprintf (_T("\nStatus from QueryServiceStatus"));
        _tprintf (_T("\nSerice Status"));
        _tprintf (_T("\ServiceType: %d"), sStatus.dwServiceType);
        _tprintf (_T("\CurrentState: %d"), sStatus.dwCurrentState);
        _tprintf (_T("\ControlsAccd: %d"), sStatus.dwControlsAccepted);
        _tprintf (_T("\Win32ExitCode: %d"), sStatus.dwWin32ExitCode);
        _tprintf (_T("\ServiceSpecificExitCode: %d"),
              sStatus.dwServiceSpecificExitCode);
        _tprintf (_T("\CheckPoint: %d"), sStatus.dwCheckPoint);
        _tprintf (_T("\ndwWaitHint: %d"), sStatus.dwWaitHint);

    }
    if (hSc != NULL) CloseServiceHandle (hSc);
    return 0;
}
```

Run 13–3 shows SimpleService operation.



```
C:\WSP4_Examples\run8>ServiceShell

Windows Service Mangement
SM$create SimpleService SimpleService SimpleService.exe

SM$start SimpleService

SM$control SimpleService interrogate

Status from QueryServiceStatus
Serice Status
dwServiceType: 16
dwCurrentState: 4
dwControlsAccepted: 7
dwWin32ExitCode: 0
dwServiceSpecificExitCode: 0
dwCheckPoint: 0
dwWaitHint: 0
SM$control SimpleService stop

SM$delete SimpleService

SM$quit

C:\WSP4_Examples\run8>
```

Run 13–3 ServiceShell: Managing Services

Sharing Kernel Objects with a Service

There can be situations in which a service and applications share a kernel object. For example, the service might use a named mutex to protect a shared memory region used to communicate with applications. Furthermore, in this example, the file mapping would also be a shared kernel object.

There is a difficulty caused by the fact that applications run in a security context separate from that of services, which can run under the system account. Even if no protection is required, it is not adequate to create and/or open the shared kernel objects with a NULL security attribute pointer (see Chapter 15). Instead, a non-NULL discretionary access control list is required at the very least—that is, the applications and the service need to use a non-NULL security attribute structure. In general, you may want to secure the objects, and, again, this is the subject of Chapter 15.

Also notice that if a service runs under the system account, there can be difficulties in accessing resources on other machines, such as shared files, from within a service.

Notes on Debugging a Service

A service is expected to run continuously, so it must be reliable and as defect-free as possible. While a service can be attached to the debugger and event logs can be used to trace service operation, these techniques are most appropriate after a service has been deployed.

During initial development and debugging, however, it is often easier to take advantage of the service wrapper presented in Program 13–2, which allows operation as either a service or a stand-alone application based on the command line `-c` option.

- Develop the "preservice" version as a stand-alone program. `serverSK`, for example, was developed in this way.
- Instrument the program with event logging or a log file.
- Once the program is judged to be ready for deployment as a service, run it without the `-c` command line option so that it runs as a service.
- Additional testing on a service is essential to detect both additional logic errors and security issues. Services can run under the system account and do not, for instance, necessarily have access to user objects, and the stand-alone version may not detect such problems.
- Normal events and minor maintenance debugging can be performed using information in the log file or event log. Even the status information can help determine server health and defect symptoms.
- If extensive maintenance is necessary, you can debug as a normal application using the `-c` option.

Summary

Windows services provide standardized capabilities to add user-developed services to Windows computers. An existing stand-alone program can be converted to a service using the methods in this chapter.

A service can be created, controlled, and monitored using the Administrative Tools or the `ServiceShell` program presented in this chapter. The SCM controls and monitors deployed services, and there are registry entries for all services.

Looking Ahead

Chapter 14 describes asynchronous I/O, which provides two techniques that allow multiple read and write operations to take place concurrently with other processing. It is not necessary to use threads; only one user thread is required.

In most cases, multiple threads are easier to program than asynchronous I/O, and thread performance is generally superior. However, asynchronous I/O is essential to the use of I/O completion ports, which are extremely useful when building scalable servers that can handle large numbers of clients.

Chapter 14 also describes waitable timers.

Additional Reading

Kevin Miller's *Professional NT Services* thoroughly covers the subject. Device drivers and their interaction with services were not covered in this chapter; a book such as Walter Oney's *Programming the Microsoft Windows Driver Model, Second Edition*, can provide that information.

Exercises

13–1. Modify Program 13–2 (`SimpleService`) to use Windows events instead of a log file. The principal functions to use are `RegisterEventSource`, `ReportEvent`, and `DeregisterEventSource`, all described in MSDN. Also consider using Vista event logging. Alternatively, use an open source logging system such as NLog (http://nlog-project.org/home).

13–2. Extend `serviceSK` to accept pause controls in a meaningful way. As suggested behavior for a paused service, it should maintain existing connections but not accept new connections. Furthermore, it should complete and respond to requests that are currently being processed, but it should not accept any more client requests.

13–3. `ServiceShell`, when interrogating service status, simply prints out the numbers. Extend it so that status is presented in a more readable form.

13–4. Convert `serverNP` (Program 12–3) into a service.

13–5. Test `serviceSK` in the *Exercises* file. Modify `serviceSK` so that it uses event logging.

This page intentionally left blank