Status will also be used in several other places to set different values, informing the SCM of the service's current status. A later section and Table 13–3 describe the valid status values in addition to SERVICE_START_PENDING.

The service control handler must set the status every time it is called, even if there is no status change.

Furthermore, any of the service's threads can call SetServiceStatus at any time to report progress, errors, or other information, and services frequently have a thread dedicated to periodic status updates. The time period between status update calls is specified in a member field in a data structure parameter. The SCM can assume an error has occurred if a status update does not occur within this time period.

```
BOOL SetServiceStatus (
   SERVICE_STATUS_HANDLE hServiceStatus,
   LPSERVICE_STATUS lpServiceStatus)
```

Parameters

hServiceStatus is the SERVICE_STATUS_HANDLE returned by RegisterServiceCtrlHandlerEx. The RegisterServiceCtrlHandlerEx call must therefore precede the SetServiceStatus call.

lpServiceStatus, pointing to a SERVICE_STATUS structure, describes service properties, status, and capabilities.

The SERVICE_STATUS Structure

The SERVICE_STATUS structure definition is:

```
typedef struct _SERVICE_STATUS {
   DWORD dwServiceType;
   DWORD dwCurrentState;
   DWORD dwControlsAccepted;
   DWORD dwWin32ExitCode;
   DWORD dwServiceSpecificExitCode;
   DWORD dwCheckPoint;
   DWORD dwWaitHint;
} SERVICE_STATUS, *LPSERVICE_STATUS;
```

Parameters

dwWin32ExitCode is the normal thread exit code for the logical service. The service must set this to NO_ERROR while running and on normal termination. Despite the name, you can use this field on 64-bit applications; there will be "32" references in other nSames.

dwServiceSpecificExitCode can be used to indicate an error while the service is starting or stopping, but this value will be ignored unless dwWin32ExitCode is set to ERROR_SERVICE_SPECIFIC_ERROR.

dwCheckPoint should be incremented periodically by the service to report its progress during all steps, including initialization and shutdown. This value is invalid and should be 0 if the service does not have a start, stop, pause, or continue operation pending.

dwWaitHint, in milliseconds, is the elapsed time between calls to SetServiceStatus with either an incremented value of dwCheckPoint value or a change in dwCurrentState. As mentioned previously, the SCM can assume that an error has occurred if this time period passes without such a SetServiceStatus call.

The remaining SERVICE_STATUS members are now described in individual sections.

Service Type

dwServiceType must be one of the values described in Table 13–1.

Table 13–1 Service Types

Value	Meaning
SERVICE_WIN32_OWN_PROCESS	Indicates that the Windows service runs in its own process with its own resources. *Program 13–2 uses this value.*
SERVICE_WIN32_SHARE_PROCESS	Indicates a Windows service that shares a process with other services, consolidating several services into a single process, which can reduce overall resource requirements.
SERVICE_KERNEL_DRIVER	Indicates a Windows device driver and is reserved for system use.
SERVICE_FILE_SYSTEM_DRIVER	Specifies a Windows file system driver and is also reserved for system use.
SERVICE_INTERACTIVE_PROCESS	This flag can be combined with only the two SERVICE_WIN32_X values. However, interactive services pose a security risk and should not be used.

Table 13–2 Service State Values

Value	Meaning
SERVICE_STOPPED	The service is not running.
SERVICE_START_PENDING	The service is in the process of starting but is not yet ready to respond to requests. For example, the worker threads have not yet been started.
SERVICE_STOP_PENDING	The service is stopping but has not yet completed shutdown. For example, a global shutdown flag may have been set, but the worker threads have not yet responded.
SERVICE_RUNNING	The service is running.
SERVICE_CONTINUE_PENDING	The service is in the process of resuming from the pause state, but it is not yet running.
SERVICE_PAUSE_PENDING	The service pause is in process, but the service is not yet safely in the pause state.
SERVICE_PAUSED	The service is paused.

Table 13–3 Controls That a Service Accepts (Partial List)

Value	Meaning
SERVICE_ACCEPT_STOP	Enables SERVICE_CONTROL_STOP.
SERVICE_ACCEPT_PAUSE_CONTINUE	Enables SERVICE_CONTROL_PAUSE and SERVICE_CONTROL_CONTINUE.
SERVICE_ACCEPT_SHUTDOWN (The ControlService function cannot send this control code.)	Notifies the service when system shutdown occurs. This enables the system to send a SERVICE_CONTROL_SHUTDOWN value to the service. For Windows system use only.
SERVICE_ACCEPT_PARAMCHANGE	The startup parameters can change without restarting. The notification is SERVICE_CONTROL_PARAMCHANGE.

For our purposes, the type is almost always SERVICE_WIN32_OWN_PROCESS, and SERVICE_WIN32_SHARE_PROCESS is the only other value suitable for user-mode services. Showing the different values, however, does indicate that services play many different roles.

Service State

dwCurrentState indicates the current service state. Table 13–2 shows the different possible values.

Controls Accepted

`dwControlsAccepted` specifies the control codes that the service will accept and process through its service control handler (see the next section). Table 13–3 enumerates three values used in a later example, and the appropriate values should be combined by bit-wise "or" (`|`). See the MSDN entry for `SERVICE_STATUS` for the three additional values.

Service-Specific Code

Once the handler has been registered and the service status has been set to `SERVICE_START_PENDING`, the service can initialize itself and set its status again. In the case of converting `serverSK`, once the sockets are initialized and the server is ready to accept clients, the status should be set to `SERVICE_RUNNING`.

The Service Control Handler

The service control handler, the callback function specified in `RegisterServiceCtrlHandlerEx`, has the following form:

```
DWORD WINAPI HandlerEx (
   DWORD dwControl,
   DWORD dwEventType,
   LPVOID lpEventData,
   LPVOID lpContext)
```

The `dwControl` parameter indicates the actual control signal sent by the SCM that should be processed.

There are 14 possible values for `dwControl`, including the controls mentioned in Table 13–3. Five control values of interest in the example are listed here:

`SERVICE_CONTROL_STOP`

`SERVICE_CONTROL_PAUSE`

`SERVICE_CONTROL_CONTINUE`

`SERVICE_CONTROL_INTERROGATE`

`SERVICE_CONTROL_SHUTDOWN`

User-defined values in the range 128–255 are also permitted but will not be used here.

dwEventType is usually 0, but nonzero values are used for device management, which is out of scope for this book. lpEventData provides additional data required by some of these events.

Finally, lpContext is user-defined data passed to RegisterServiceCtrlHandlerEx when the handler was registered.

The handler is invoked by the SCM in the same thread as the main program, and the function is usually written as a switch statement. This is shown in the examples.

Event Logging

Services run "headless" without user interaction, so it is not generally appropriate for a service to display status messages directly. Prior to Vista and NT6, some services would create a console, message box, or window for user interaction; those techniques are no longer available.

The solution is to log events to a log file or use Windows event logging functionality. Such events are maintained within Windows and can be viewed from the event viewer provided in the control panel's Administrative Tools.

The upcoming SimpleService example (Program 13–2) logs significant service events and errors to a log file; an exercise asks you to modify the program to use Windows events.

Example: A Service "Wrapper"

Program 13–2 performs the conversion of an arbitrary _tmain to a service. The conversion to a service depends on carrying out the tasks we've described. The existing server code (that is, the old _tmain function) is invoked as a thread or process from the function ServiceSpecific. Therefore, the code here is essentially a wrapper around an existing server program.

The command line option -c specifies that the program is to run as a standalone program, perhaps for debugging. Without the option, there is a call to StartServiceCtrlDispatcher.

Another addition is a log file; the name is hard-coded for simplicity. The service logs significant events to that file. Simple functions to initialize and close the log and to log messages are at the end.

Several other simplifications and limitations are noted in the comments.

Program 13–2 `SimpleService`: A Service Wrapper

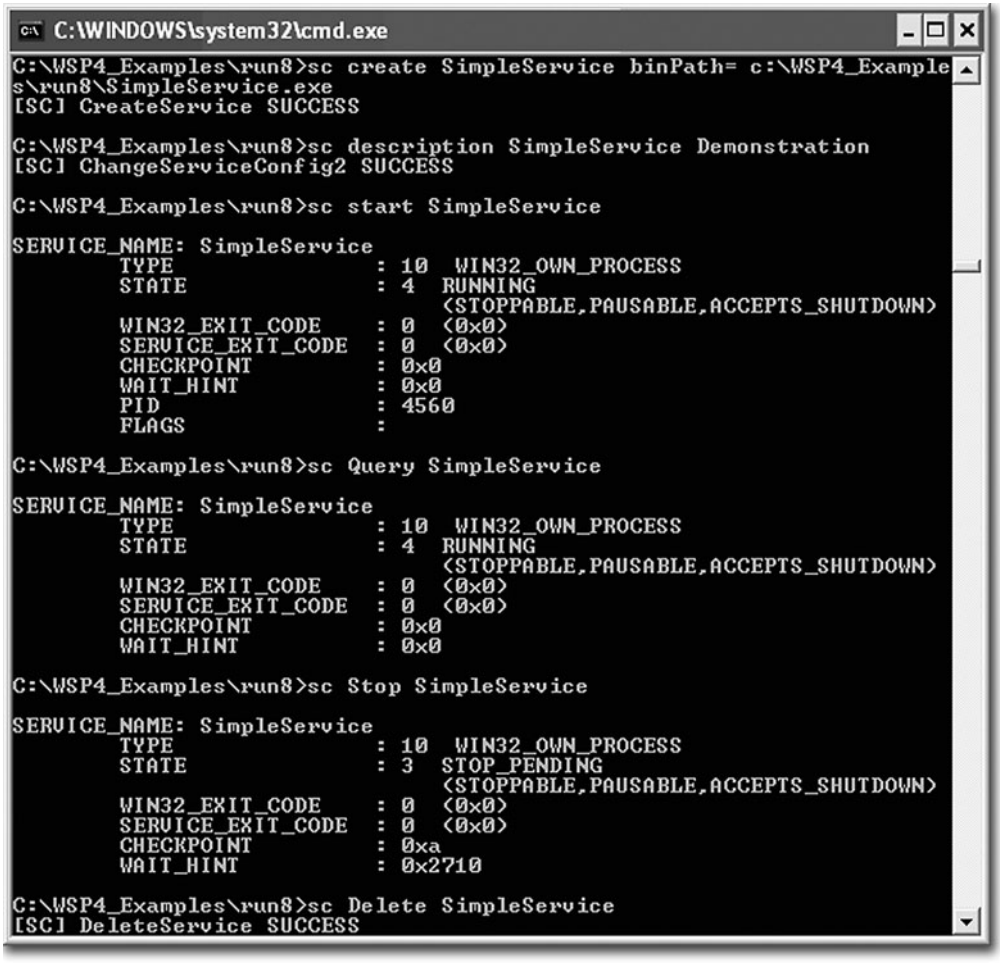
```
/*Chapter 13. SimpleService.c
   Simplest example of a Windows Service
   All it does is update the checkpoint counter
   and accept basic controls.
   You can also run it as a stand-alone application. */

#include "Everything.h"
#include <time.h>
#define UPDATE_TIME 1000/* One second between updates */

VOID LogEvent (LPCTSTR, WORD), LogClose();
BOOL LogInit(LPTSTR);
void WINAPI ServiceMain (DWORD argc, LPTSTR argv[]);
VOID WINAPI ServerCtrlHandler(DWORD);
void UpdateStatus (int, int);
int  ServiceSpecific (int, LPTSTR *);

static BOOL shutDown = FALSE, pauseFlag = FALSE;
static SERVICE_STATUS hServStatus;
static SERVICE_STATUS_HANDLE hSStat; /* handle for setting status */

static LPTSTR serviceName = _T("SimpleService");
static LPTSTR logFileName = _T(".\\LogFiles\\SimpleServiceLog.txt");
static BOOL consoleApp = FALSE, isService;

/*  Main routine that starts the service control dispatcher */
/* Optionally, run as a stand-alone console program*/
/* Usage: simpleService [-c]             */
/*        -c says to run as a console app, not a service*/

VOID _tmain (int argc, LPTSTR argv[])
{
   SERVICE_TABLE_ENTRY DispatchTable[] =
   {
      { serviceName,      ServiceMain},
      { NULL,                NULL }
   };

   Options (argc, argv, _T("c"), &consoleApp, NULL);
   isService = !consoleApp;
   /* Initialize log file */
   if (!LogInit (logFileName)) return;

   if (isService) {
      LogEvent(_T("Starting Dispatcher"), EVENTLOG_SUCCESS);
      StartServiceCtrlDispatcher (DispatchTable);
   } else {
```

```
        LogEvent(_T("Starting application"), EVENTLOG_SUCCESS);
        ServiceSpecific (argc, argv);
    }
    LogClose();
    return;
}

/* ServiceMain entry point, called by main program.  */
void WINAPI ServiceMain (DWORD argc, LPTSTR argv[])
{
    LogEvent (_T("Entering ServiceMain."), EVENTLOG_SUCCESS);

    hServStatus.dwServiceType = SERVICE_WIN32_OWN_PROCESS;
    hServStatus.dwCurrentState = SERVICE_START_PENDING;
    hServStatus.dwControlsAccepted = SERVICE_ACCEPT_STOP |
        SERVICE_ACCEPT_SHUTDOWN | SERVICE_ACCEPT_PAUSE_CONTINUE;
    hServStatus.dwWin32ExitCode = NO_ERROR;
    hServStatus.dwServiceSpecificExitCode = 0;
    hServStatus.dwCheckPoint = 0;
    hServStatus.dwWaitHint = 2 * UPDATE_TIME;

    hSStat =
        RegisterServiceCtrlHandler( serviceName, ServerCtrlHandler);

    if (hSStat == 0) {
        LogEvent (_T("Cannot register handler"), EVENTLOG_ERROR_TYPE);
        hServStatus.dwCurrentState = SERVICE_STOPPED;
        hServStatus.dwWin32ExitCode = ERROR_SERVICE_SPECIFIC_ERROR;
        hServStatus.dwServiceSpecificExitCode = 1;
        UpdateStatus (SERVICE_STOPPED, -1);
        return;
    }

    LogEvent (_T("Control handler registered"), EVENTLOG_SUCCESS);
    SetServiceStatus (hSStat, &hServStatus);
    LogEvent (_T("Status SERVICE_START_PENDING"), EVENTLOG_SUCCESS);

    /*  Start service-specific work; the generic work is complete */
    ServiceSpecific (argc, argv);

    /*  only return here when the ServiceSpecific function
        completes, indicating system shutdown. */
    LogEvent (_T("Service threads shut down"), EVENTLOG_SUCCESS);
    LogEvent (_T("Set SERVICE_STOPPED status"), EVENTLOG_SUCCESS);
    UpdateStatus (SERVICE_STOPPED, 0);
    LogEvent (_T("Status set to SERVICE_STOPPED"), EVENTLOG_SUCCESS);
    return;
}
```

```
/* service-specific function, or "main"; called from ServiceMain */
int ServiceSpecific (int argc, LPTSTR argv[])
{
   UpdateStatus (-1, -1); /* change to status; increment checkpoint */
   /* Start the server as a thread or process */
   /* Assume the service starts in 2 seconds. */
   UpdateStatus (SERVICE_RUNNING, -1);
   LogEvent (_T("Status update. Service running"), EVENTLOG_SUCCESS);

   /* Update the status periodically. */
   /*** The update loop could be on a separate thread ***/
   /* Also, check the pauseFlag - See Exercise 13-1 */
   LogEvent (_T("Starting main service loop"), EVENTLOG_SUCCESS);
   while (!shutDown) { /* shutDown is set on a shutDown control */
      Sleep (UPDATE_TIME);
      UpdateStatus (-1, -1);  /* Assume no change */
      LogEvent (_T("Status update. No change"), EVENTLOG_SUCCESS);
   }
   LogEvent (_T ("Server process has shut down."), EVENTLOG_SUCCESS);
   return 0;
}

/* Control Handler Function */
VOID WINAPI ServerCtrlHandler( DWORD dwControl)
{
   switch (dwControl) {
   case SERVICE_CONTROL_SHUTDOWN:
   case SERVICE_CONTROL_STOP:
      shutDown = TRUE;/* Set the global shutDown flag */
      UpdateStatus (SERVICE_STOP_PENDING, -1);
      break;
   case SERVICE_CONTROL_PAUSE:
      pauseFlag = TRUE;
      /* Pause implementation is Exercise 13-1 */
      break;
   case SERVICE_CONTROL_CONTINUE:
      pauseFlag = FALSE;
      /* Continue is also an exercise */
      break;
   case SERVICE_CONTROL_INTERROGATE:
      break;
   default:
      if (dwControl > 127 && dwControl < 256) /* User Defined */
      break;
   }
   UpdateStatus (-1, -1);
   return;
}
```

```
void UpdateStatus (int NewStatus, int Check)
/*  Set service status and checkpoint (specific value or increment) */
{
   if (Check < 0 ) hServStatus.dwCheckPoint++;
   else      hServStatus.dwCheckPoint = Check;

   if (NewStatus >= 0) hServStatus.dwCurrentState = NewStatus;
   if (isService) {
      if (!SetServiceStatus (hSStat, &hServStatus)) {
         LogEvent (_T("Cannot set status"), EVENTLOG_ERROR_TYPE);
         hServStatus.dwCurrentState = SERVICE_STOPPED;
         hServStatus.dwWin32ExitCode = ERROR_SERVICE_SPECIFIC_ERROR;
         hServStatus.dwServiceSpecificExitCode = 2;
         UpdateStatus (SERVICE_STOPPED, -1);
         return;
      } else {
         LogEvent (_T("Service Status updated."), EVENTLOG_SUCCESS);
      }
   } else {
      LogEvent (_T("Stand-alone status updated."), EVENTLOG_SUCCESS);
   }
   return;
}

/* Simple file based event logging */
static FILE * logFp = NULL;
/* Very primitive logging service, using a file */
VOID LogEvent (LPCTSTR UserMessage, WORD type)
{
   TCHAR cTimeString[30] = _T("");
   time_t currentTime = time(NULL);
   _tcsncat (cTimeString, _tctime(&currentTime), 30);
   /* Remove the new line at the end of the time string */
   cTimeString[_tcslen(cTimeString)-2] = _T('\0');
   _ftprintf(logFp, _T("%s. "), cTimeString);
   if (type == EVENTLOG_SUCCESS || type == EVENTLOG_INFORMATION_TYPE)
      _ftprintf(logFp, _T("%s"), _T("Information. "));
   else if (type == EVENTLOG_ERROR_TYPE)
      _ftprintf(logFp, _T("%s"), _T("Error.       "));
   else if (type == EVENTLOG_WARNING_TYPE)
      _ftprintf(logFp, _T("%s"), _T("Warning.     "));
   else
      _ftprintf(logFp, _T("%s"), _T("Unknown.     "));

   _ftprintf(logFp, _T("%s\n"), UserMessage);
   fflush(logFp);
   return;
}
```

```
BOOL LogInit(LPTSTR name)
{
   logFp = _tfopen (name, _T("a+"));
   if (logFp != NULL) LogEvent (_T("Initialized Logging"),
         EVENTLOG_SUCCESS);
   return (logFp != NULL);
}

VOID LogClose()
{
   LogEvent (_T("Closing Log"), EVENTLOG_SUCCESS);
   return;
}
```

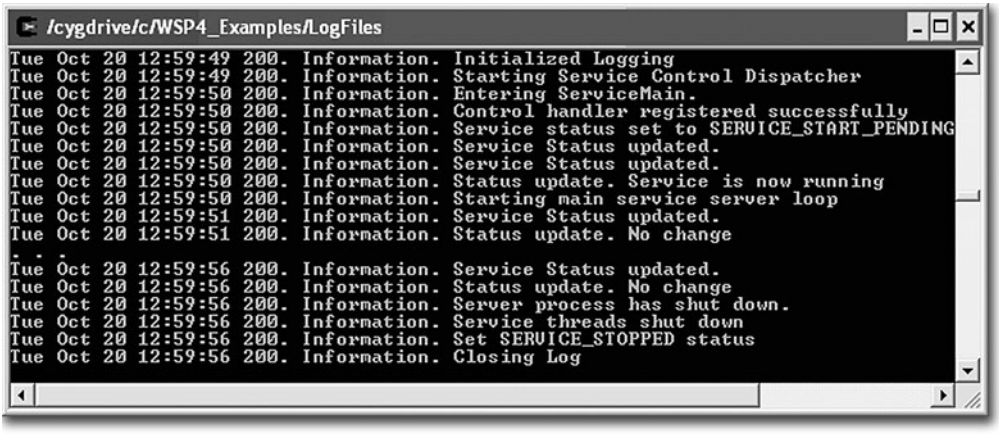


Run 13–2a `SimpleService`: Controlled by `sc`

Running the Simple Service

Run 13–2a shows the `sc` command tool creating, starting, querying, stopping, and deleting `SimpleService`. Only an administrator can perform these steps.

Run 13–2b shows the log file.



Run 13–2b `SimpleServiceLog.txt`: The Log File

Managing Windows Services

Once a service has been written, the next task is to put the service under the control of the SCM so that the SCM can start, stop, and otherwise control the service. While `sc.exe` and the Services administrative tool can do this, you can also manage services programmatically, as we'll do next.

There are several steps to open the SCM, create a service under SCM control, and then start the service. These steps do not directly control the service; they are directives to the SCM, which in turn controls the specified service.

Opening the SCM

A separate process, running as "Administrator," is necessary to create the service, much as `JobShell` (Chapter 6) starts jobs. The first step is to open the SCM, obtaining a handle that then allows the service creation.

```
SC_HANDLE OpenSCManager (
   LPCTSTR lpMachineName,
   LPCTSTR lpDatabaseName,
   DWORD dwDesiredAccess)
```

Parameters

lpMachineName is NULL if the SCM is on the local computer, but you can also access the SCM on networked machines.

lpDatabaseName is also normally NULL.

dwDesiredAccess is normally SC_MANAGER_ALL_ACCESS, but you can specify more limited access rights, as described in the on-line documentation.

Creating and Deleting a Service

Call CreateService to register a service.

```
SC_HANDLE CreateService (
   SC_HANDLE hSCManager,
   LPCTSTR lpServiceName,
   LPCTSTR lpDisplayName,
   DWORD dwDesiredAccess,
   DWORD dwServiceType,
   DWORD dwStartType,
   DWORD dwErrorControl,
   LPCTSTR lpBinaryPathName,
   LPCTSTR lpLoadOrderGroup,
   LPDWORD lpdwTagId,
   LPCTSTR lpDependencies,
   LPCTSTR lpServiceStartName,
   LPCTSTR lpPassword);
```

As part of CreateService operation, new services are entered into the registry under:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services
```

Do not, however, attempt to bypass CreateService by manipulating the registry directly; we just point this out to indicate how Windows keeps service information.

Parameters

hSCManager is the SC_HANDLE obtained from OpenSCManager.