Creating NT Services

by John Chaytor

This article shows how you can create services to run under Windows NT. Services can be useful on both a single workstation PC and in a network environment. If you look at `Settings/Control Panel/ Services` you will be able to see a list of services defined on your machine. Examples of services that run on my machine are: FTP, Gopher and WWW publishing, telephony, remote access, event logging and services supporting my video card.

Don't confuse the term *service* with server in the client/server architecture. A Win32 service can indeed be part of a client/server relationship (SQL Server is an example of such a service) but it can equally be a stand-alone utility. Two examples of the latter are automated backup and disk de-fragment utilities such as Diskeeper.

So what is so special about a service? It is an executable program but it runs under the control of the Service Control Manager (SCM). It needs to be defined in the registry. The SCM controls when the service starts and ends, either from information in the registry (in response to user requests from the `Control Panel/Services` applet) or via API calls from applications. It can be started automatically at system boot time before any user logs on.

Services do not need a user to be logged on in order to execute, but they can, if required, run under a specific user id account. A service accessing a database may need a valid user id to enable it to connect to the DBMS. Services remain running when a user logs off.

Services can have dependency information stored in the registry of services that need to be started before starting the current service. The SCM will start them in the correct sequence.

Services are usually linked as console applications. They must have two types of entry points declared with the `stdcall` directive: a `ServiceMain` procedure which is called on service start-up and a `ControlHandler` procedure which is called to query or change the status of the service once it is running.

A single program (process) can contain the code for multiple services. This is useful if you have services that need to communicate. By putting them in the same process you avoid the overhead of inter-process communication. However, if your service required a special logon-id the service must run in its own process.

So far so good, but what are the drawbacks? Services can be difficult to debug. I would recommend that you develop the main logic of your service as a standard console or basic GUI app then convert it to a service once you are happy with it. Unlike standard console apps, in a service you cannot write to the console via `WriteLn` (there may not be any users logged on to the machine). You must ensure that the id that the service runs under (`localsystem` by default) has the correct access privileges for all the required resources. It may be necessary to set up a special userid and password specifically for your service. If you do this, remember to set the `password never expires` option to avoid problems in 30 days or so!

If you access network drives from the service, ensure that you use the UNC paths to access the drives (eg `\\Admin01\C:\Payroll\Files\Wages.DB`). You cannot rely on mapping drive letters to a network drive, as these may be invalid.

Getting Started

Initially we will create a very simple 'service' that just beeps periodically to show that it is running, to allow us to explore the interaction between the SCM and a service (the delay period can be altered at service start-up). It will accept start, pause, resume and stop requests. To aid debugging we'll add the capability to write to the event log (this can be used by standard applications also). Once we are happy with that I'll describe the two classes I developed to encapsulate the interface to the SCM which hides the service start-up and control mechanism allowing you to concentrate on the functionality of your service.

Delphi 2 does not come with the Pascal version of the header file WinSvc.H. I have supplied my version of this, called WINSVCX.PAS, (so it won't conflict with the official version supplied with Delphi 3) on the disk. It only contains the definitions required for this article.

Service Program Structure

Listing 1 shows the source for the first demo program (on the disk as DEMOSV1.DPR) minus error checking and event logging. This program contains a single service. There are special considerations required for shared service processes and these are highlighted as they arise.

Although the program is usually executed by the SCM it can also be executed at the command line like any regular console program. We can use this fact to our advantage. When the SCM starts the program it uses an entry in the registry (see later) to get the fully qualified path name for the EXE file: the services described here do not define a parameter. Hence, if the program is started without parameters it can assume that it is being started by the SCM. If the program is passed a parameter this means that a user started the program at the command line. In the demo programs provided I use this fact to automatically install or uninstall the service by passing a parameter of `install` or `uninstall`. This idea

came from examples in the MSDN and avoids the need to use the SC utility or manually edit the registry. See the file SERVICES.TXT on the disk for further MSDN references.

As well as the main entry point, a service program must have two additional entry points. `ServiceMain` is the entry point for the main thread of the service. A shared service process may have more than one of these entry points or it may use the single entry point for all services. I use a single entry point (see `DemoSv2`). `ControlHandler` is the entry point called to update or query the running status of the service. Each service must have its own unique entry point.

Process Start-Up

After the SCM starts the process it waits for the process to call the `StartServiceCtrlDispatcher` function to register all the services it contains. A single service process (such as `DemoSv1`) should call this immediately as any service initialisation can be done when `ServiceMain` is called by the SCM. A shared service process should do any process-wide initialisation first (ie things that need to be done before any service starts) before it calls `StartServiceCtrlDispatcher`. However, if this initialisation is going to take over 30 seconds the SCM would timeout and assume that an error has occurred. To get round this you would need to create an extra thread to perform the process wide initialisation and call `StartServiceCtrlDispatcher` within the timeout period. You'd then need to implement a mechanism to inform the services that process initialisation has completed before they are allowed to start.

Note that if you accidentally start the program yourself from the command line without passing parameters, the call to `StartServiceCtrlDispatcher` will timeout as the SCM is not expecting the call. No damage will be done though.

The program will only be started by the SCM when the first service needs to be started. If additional services are to be started in the same process the SCM will simply call the relevant `ServiceMain` entry point for the service being started as specified in the table passed to `StartServiceCtrlDispatcher`.

➤ *Listing 1*

```
unit DemoSv1;
Uses
  Windows, SysUtils, Registry, WinSvcX, Demo1Log;
const
  DemoServiceName        = 'DemoService1';
  DemoServiceDisplayName = 'Demonstration service 1';
  FTerminated: Boolean;
  FServiceStatus: TServiceStatus;
  FServicStatusHandle: SERVICE_STATUS_HANDLE;
procedure DemoServiceHandler(Code: Integer); StdCall;
begin
  case code of
    SERVICE_CONTROL_STOP:
      begin
        With FServiceStatus do begin
          dwCurrentState := SERVICE_STOP_PENDING;
          dwWin32ExitCode := 0;
          dwServiceSpecificExitCode := 0;
        end;
      end;
    SERVICE_CONTROL_PAUSE:
      FServiceStatus.dwCurrentState := SERVICE_PAUSED;
    SERVICE_CONTROL_CONTINUE:
      FServiceStatus.dwCurrentState := SERVICE_RUNNING;
  end;
  SetServiceStatus(FServicStatusHandle,FServiceStatus);
  if FServiceStatus.dwCurrentState = SERVICE_STOP_PENDING
  then
    FTerminated := True;
end;
Procedure DemoServiceMain(NumArgs: DWord; Args: PCharArray);
  StdCall;
var
  InitialisedOK: Boolean;
  BeepDelay: Integer;
begin
  BeepDelay := 1000;
  FServicStatusHandle := RegisterServiceCtrlHandler(
    DemoServiceName,@DemoServiceHandler);
  if FServicStatusHandle <> 0 then begin
      FillChar(FServiceStatus,sizeof(TServiceStatus),0);
      With FServiceStatus do begin
        dwServiceType := SERVICE_WIN32_OWN_PROCESS;
        dwCurrentState := SERVICE_START_PENDING;
        dwControlsAccepted := SERVICE_ACCEPT_STOP or
          SERVICE_ACCEPT_PAUSE_CONTINUE;
      end;
      { Set status to pending before we do our
        initialisation }
      if SetServiceStatus(FServicStatusHandle,
        FServiceStatus) then begin
        { Do initialisation here. If it takes > 1 sec you
          should call SetServiceStatus passing wait hints
          and checkpoints to show progress is being made }
        { Simulate time taken to initialise }
        Sleep(1000);
        InitialisedOK := True;
        { We assume initialisation was OK for this demo! }
        if InitialisedOK then begin
          FServiceStatus.dwCurrentState := SERVICE_RUNNING;
          if SetServiceStatus(FServicStatusHandle,
          FServiceStatus) then begin
            { Main loop of service process }
            While not FTerminated do begin
              Sleep(BeepDelay);
              if not (FServiceStatus.dwCurrentState =
                SERVICE_PAUSED) then
                MessageBeep(0);
            end;
            if FServiceStatus.dwCurrentState =
              SERVICE_STOP_PENDING then begin
              { Do cleanup processing here }
              FServiceStatus.dwCurrentState :=
                SERVICE_STOPPED;
              SetServiceStatus(FServicStatusHandle,
                FServiceStatus);
            end;
          end;
        end else With FServiceStatus do begin
          dwCurrentState := SERVICE_STOPPED;
          dwWin32ExitCode := 666;
          { Set a code to indicate reason for failure }
          SetServiceStatus(FServicStatusHandle,
            FServiceStatus);
        end;
      end;
    end;
end;
{ Main() entry point }
var
  Param: ShortString;
  ServiceEntryTable: PServiceTableEntry;
begin
  FTerminated := False;
  Param := UpperCase(ParamStr(1));
  if (Param = 'INSTALL') or (Param = 'I')  then
    InstallService
  else
    if (Param = 'UNINSTALL') or (Param = 'U') then
      UninstallService
    else
      if (Param = 'VERSION') or (Param = 'V') then
        DisplayVersion
      else
        if Param = '' then begin
          { We should have been called by the SCM,
            so connect to it }
          ServiceEntryTable :=
            AllocMem(2*SizeOf(TServiceTableEntry));
          try
            ServiceEntryTable^.lpServiceName:=
              DemoServiceName;
            ServiceEntryTable^.lpServiceProc:=
              @DemoServiceMain;
            { The CtrlDispatcher loops round waiting for
              control requests for the service(s) detailed
              in the ServiceEntryTable array. It will not
              return until the all services in the process
              terminate (or an error has occurred) }
            StartServiceCtrlDispatcher(ServiceEntryTable^);
          finally
            FreeMem(ServiceEntryTable);
          end;
        end else
          DisplaySyntaxOptions;
end.
```

The StartServiceCtrlDispatcher function is passed a dispatch table. This is a NULL delimited array of TServiceTableEntry structures: one for each supported service. This structure consists of two fields: a pointer to a NULL terminated string containing the service name and a pointer to its ServiceMain entry point. Note that as you supply the address of the entry point this allows you to call this function anything you wish (its name is not exported). In DemoSv1 it is called DemoServiceMain.

When you call StartServiceCtrlDispatcher, the function remains in a loop waiting to receive commands from the SCM to be dispatched to the service(s) in the process. Due to this, the function does not return until all services in the process have terminated.

If the call to StartServiceCtrlDispatcher succeeds it will start the required service. To do this, the dispatcher creates a thread and calls the ServiceMain entry point supplied in the dispatch table. This is an important point. We do not need to create a thread for the service. The dispatcher does this for us.

dwServiceType	SERVICE_WIN32_OWN_PROCESS (there is only one service in this process) or SERVICE_WIN32_SHARE_PROCESS (there is more than one services in the process).
dwCurrentState	SERVICE_STOPPED or SERVICE_START_PENDING (this is the initial value) or SERVICE_STOP_PENDING or SERVICE_RUNNING or SERVICE_CONTINUE_PENDING or SERVICE_PAUSE_PENDING or SERVICE_PAUSED.
dwControlsAccepted	A combination of the flags SERVICE_ACCEPT_STOP, SERVICE_ACCEPT_PAUSE_CONTINUE and SERVICE_ACCEPT_SHUTDOWN. This lets the SCM know what instructions can be passed to the service. These can be changed during the life of the service.
dwWin32ExitCode	The two ExitCode fields are used to pass the return code back to the SCM when service stops.
dwServiceSpecificExitCode	
dwCheckPoint	See *Lengthy operations* in the article.
dwWaitHint	See *Lengthy operations* in the article.

➤ *Table 1: TServiceStatus fields*

```
TServiceStatus = Record
    dwServiceType: Integer;
    dwCurrentState: Integer;
    dwControlsAccepted: Integer;
    dwWin32ExitCode: Integer;
    dwServiceSpecificExitCode: Integer;
    dwCheckPoint: Integer;
    dwWaitHint: Integer;
  End;
```

➤ *Listing 2*

Service Start-Up

The DemoServiceMain procedure is passed two parameters. The first is the number of arguments and the second is a pointer to an array of PChars (the arguments). The first argument is always the name of the service being started. This allows a single entry point to be common to multiple services (more on this later). Any optional parameters which follow are the start-up parameters typed on the Control Panel Services applet by the user. This information can be used by the service in any way you wish. DemoSv1 uses the last parameter to specify the delay between beeps.

In the DemoServiceMain procedure the first thing that needs to be done is to call the RegisterServiceCtrlHandler function. This expects two parameters: a pointer to the service name and the entry point of the Handler procedure described earlier. This procedure is called DemoControlHandler in DemoSv1. It will be called from the dispatcher whenever there is a request to change the status of the service (eg pause, restart or stop) or interrogate its current state. The handle returned from RegisterServiceCtrlHandler will be unique for the service. This needs to be stored away, as it must be used when informing the SCM of the status of the service.

After registering the Handler procedure, the ServiceMain procedure needs to inform the SCM of its current status. This is done via a call to SetServiceStatus. This function accepts the service handle (returned from RegisterServiceCtrlHandler) and a pointer to a TServiceStatus structure. Listing 2 shows the format of this structure and Table 1 shows the meaning of the fields.

At this point we would start doing processing specific to our service (eg waiting on a TCP port, setup timers for scheduling etc).

In the DemoSv1 example it simply goes into a loop and beeps each time round. Not very exciting! However, being a service, it will stay executing if you log off the machine, beeping away to itself. While in its process loop, the service checks two fields: FTerminated, to see if it should stop processing, and the contents of dwCurrentState to see if the service has changed its status to paused.

These states are set by the code in the ServiceHandler. If you remember, this is called by the SCM when it needs to query or change the state of the service. If it requests that the service be stopped, paused or resumed the code simply updates dwCurrentState to indicate this. It does not suspend or resume the service thread! If you did this you may pause the service in the middle of processing a request. In the examples it is up to the main service to respond to these requests. For example, if a

Operation	Initial status	Final status
Start	SERVICE_START_PENDING	SERVICE_RUNNING
Stop	SERVICE_STOP_PENDING	SERVICE_STOPPED
Pause	SERVICE_PAUSE_PENDING	SERVICE_PAUSED
Continue	SERVICE_CONTINUE_PENDING	SERVICE_RUNNING

➤ *Table 2: Pending states*

```
procedure InstallService;
var
  hSCManager: SC_Handle;
  hService: SC_Handle;
Begin
  hSCManager:= OpenSCManager(nil,nil,SC_MANAGER_ALL_ACCESS);
  If hSCManager <> 0 then
    try
      hService:= CreateService(hSCManager,DemoServiceName,DemoServiceDisplayName,
        SERVICE_ALL_ACCESS,SERVICE_WIN32_OWN_PROCESS,
        SERVICE_DEMAND_START,SERVICE_ERROR_NORMAL,
        PChar(ParamStr(0)),nil,nil,nil,nil,nil);
      if hService <> 0 then begin
        WriteLn('Service was installed successfully.');
        AddEventDetailsToRegistry;
      end else
        WriteLn(Format('Failed to create the service. Error was ''%s''',
          [SysErrorMessage(GetLastError)]));
    finally
      CloseServiceHandle(hSCManager)
    end else
      WriteLn(Format('Failed to open Service Control Manager. Error was ''%s''',
        [SysErrorMessage(GetLastError)]));
End;
procedure UninstallService;
Var
  hSCManager: SC_Handle;
  hService: SC_Handle;
Begin
  hSCManager := OpenSCManager(nil,nil,SC_MANAGER_ALL_ACCESS);
  If hSCManager <> 0 then
    try
      hService := OpenService(hSCManager,DemoServiceName,SERVICE_ALL_ACCESS);
      if hService <> 0 then
        try
          if DeleteService(hService) then begin
            WriteLn('Service was uninstalled successfully.');
            RemoveEventDetailsFromRegistry;
          end else
            WriteLn(Format('Failed to delete service. Error was ''%s''',
              [SysErrorMessage(GetLastError)]));
        finally
          CloseServiceHandle(hService);
        end else
          WriteLn(Format('Failed to open service "%s": Error was ''%s''',
            [DemoServiceName, SysErrorMessage(GetLastError)]));
    finally
      CloseServiceHandle(hSCManager)
    end else
      WriteLn(Format('Failed to open Service control Manager. Error was ''%s''',
        [SysErrorMessage(GetLastError)]));
End;
```

➤ *Listing 3*

service is paused, it should honour all requests that it is currently servicing then refuse to process any more requests. The examples do this by checking the current service state.

The code parameter passed to `DemoServiceHandler` indicates what the SCM is requesting. The handler consists of a single case statement specifying each option it is expecting from the SCM. Regardless of why the SCM called this routine it should always call `SetServiceStatus` to update the SCM. If we are passed a request to pause or resume the service the routine simply updates the status. The service thread must react to this. When the SCM requests that the service be stopped, the handler sets the status to stop pending. It then sets the `FTerminated` flag to `True`. At the end of the `case` statement the `SetServiceStatus` call will update the SCM to 'stop pending.' The service thread checks this field periodically to see if it should terminate. At this point it should do all required cleanup. It then needs to set the status to `SERVICE_STOPPED` and inform the SCM via a call to `SetServiceStatus`. If the service fails to respond promptly the SCM will return an error.

If this is the last service to stop (it is in `DemoSv1`) the original call to `StartServiceCtrlDispatcher` returns and the process terminates.

Service Configuration Data

If you need to store information for your service it is recommended that you store it under the unique key in

```
HKEY_LOCAL_MACHINE\SYSTEM\
  CurrentControlSet\Services\
  NameOfService.
```

Lengthy Operations

Start, Stop, Pause and Continue requests can potentially take some considerable time to complete successfully. As the SCM may timeout before the service has completed the request a mechanism has been defined to inform the SCM that the action is still being processed. This is done via the `dwCheckPoint` and `dwWaitHint` fields of the `TServiceStatus` structure. At the start of the operation the status needs to be set to one of the pending states shown in Table 2. Before setting the status the `dwWaitHint` should be set to the maximum time (in milliseconds) that the SCM should wait before it will be updated again with the new status. This is repeated until the operation is completed. The `dwCheckPoint` field should be incremented to show the progress of the operation. At the end of the operation the status field should be set to the value shown in Table 2. At this point, the `dwCheckPoint` field should be reset to zero. See `DemoSv2` for an example of how this could be implemented.

Installing A Service

Listing 3 shows the calls made to automatically register the `DemoSv1` service to Windows. This is done if you pass `INSTALL` or `I` as a parameter to the program. There are essentially three calls required. First a call is made to `OpenSCManager` to connect to the SCM: you will need

administrator privilege for this to be successful. If this was successful a call is made to `CreateService` to create the service. The declaration for this API is shown in Listing 4, and Table 3 shows the meaning of each parameter.

Once the service entry has been created we then close the handle by calling `CloseServiceHandle`. Note the call to `AddEventDetailsToRegistry`: this is covered in the section detailing event logging.

Removing A Service

Listing 3 also shows the calls made to automatically uninstall the `DemoSv1` service from Windows. This is done if you pass `UNINSTALL` or `U` as a parameter to the program. This is essentially the reverse of the above process. A connection is made to the SCM as before. A call to `OpenService` then opens the named service. Once opened it is deleted using `DeleteService`. If the service is currently running it will be flagged for deletion. Finally the connection to the SCM is closed.

Again, notice the call to `RemoveEventDetailsFromRegistry`, this is covered in the next section.

The Event Viewer

If you look at `Start/Programs/Administrative Tools (Common)/Event Viewer` you should see a log of events that have been generated by device drivers, services, applications etc on your system for the past few days. The `Log/System` option displays events mainly generated by device drivers. It is recommended that the application log be used for services.

Writing Events To The Log

Writing events to the log is relatively easy but you have a bit of work to do before you can call the APIs.

You will need to create a message resource containing the messages you intend to use and update the registry to indicate the source of this resource file (which can be in either an EXE or a DLL) before you can start generating events. The demonstration programs on the disk update the registry for you.

hSCManager	Handle from OpenSCManager.
ServiceName	Service name.
DisplayName	Text that appears on the Control panel services applet.
DwDesiredAccess	Level of access required. Need update access.
DwServiceType	Indicates if the process runs 1 or more services.
DwStartType	When the service starts. At start-up or on demand.
DwErrorControl	What SCM should do it the service fails to start.
BinaryPathName	Fully qualified pathname to the EXE file. From ParamStr(0).
LpLoadOrderGroup	If the service is in a group which defines start order.
LpTagId	Order within previous group.
Dependencies	Names of services to start before this service.
ServiceStartName	Id of account to run service in.
Password	To use for the above.

➤ *Table 3: CreateService parameters*

```
Function CreateService(hSCManager: TSC_HANDLE; ServiceName, DisplayName: PChar;
  dwDesiredAccess, dwServiceType, dwStartType, dwErrorControl: Integer;
  BinaryPathName, lpLoadOrderGroup: PChar; lpTagId: PInteger;
  Dependencies, ServiceStartName, Password: PChar): TSC_Handle; StdCall;
```

➤ *Listing 4*

```
MessageId=
Severity=Informational
SymbolicName=DEMO1_SERVICE_STARTED
Language=English
The service started successfully.
.
MessageId=
Severity=Warning
SymbolicName=DEMO1_SERVICE_PAUSED
Language=English
The service was paused.
.
MessageId=
Severity=Informational
SymbolicName=DEMO1_SERVICE_CONTINUED
Language=English
The service was resumed after being paused for %1 milliseconds.
```

➤ *Figure 1*

Creating Message Resource

Event logging requires a special kind of resource to support internationalisation of event messages. To create this resource you will need to use the message compiler (MC.EXE) which comes with the SDK. The source text file (with a .MC extension) is fairly straightforward, Figure 1 shows part of the example for `DemoSv1`. The message compiler generates a C header, a BIN file containing the resource (MSG00001.BIN, for English messages) and an RC file (which specifies the language and source files, in this case MSG00001.BIN). The RC file needs to be compiled to create a .RES file which, in turn, needs to be linked to the program using the `$R` directive. You will need to convert the header file into

```
var
  EventSource: THandle;
  Inserts: Array[0..0] Of PChar;
begin
  EventSource := RegisterEventSource(nil,'DemoSerivce');
  Try
    Inserts[0] := '1000';
    ReportEvent(EventSource, EVENTLOG_WARNING_TYPE,0,Id,nil,1,0,Inserts,nil);
  finally
    DeRegisterEventSource(EventSource);
  end;
end;
```

➤ *Listing 5*

```
function ReportEvent(hEventLog: THandle; wType, wCategory: Word;
  dwEventID: DWORD; lpUserSid: Pointer; wNumStrings: Word; dwDataSize: DWORD;
  Inserts: PCharArray; lpRawData: Pointer): BOOL; stdcall;
```

➤ *Listing 6*

hEventLog	The handle returned from RegisterEventSource.
wType	One of the pre-defined constants for information, error or warning types.
WCategory	Can be anything meaningful for the service.
dwEventId	The ID of one of the messages in the resource created by MC.EXE.
lpUserSid	The security identifier for the user. The demos specify nil.
wNumStrings	The number of values to be used to insert into the message.
dwDataSize	The number of bytes of raw data which can be optionally added to the event record.
lpStrings	Points to an array of PChars which contain the values of the inserts.
lpRawData	Pointer to the buffer to be written to the event record.

➤ *Table 4: ReportEvent parameters*

a Pascal unit. In the examples for this article I have supplied all the intermediate files of this process in case you don't have the message compiler. Note that the messages can contain inserts (`%1`, `%2` etc) which can be specified at run time.

Updating The Registry

You can either link the message resource into your service EXE (the demos do this) or into a different program or DLL. Whichever you choose, you must update the registry to let Windows know where the file is. A new key under

```
HKEY_LOCAL_MACHINE\System\
  CurrentControlSet\
  Services\EventLog\Application
```

needs to be created (this is known as the source name). The name of this key should be unique for the service (use the service name). You then need to create two values for this key: `EventMessageFile` is the fully qualified name of the EXE or DLL containing the message resource, and `TypesSupported` contains the bit mask of message types supported (info, warning and error).

If you fail to add the source name to the registry the event will still be logged but there will be no message text displayed for the event.

In the source on the disk look for the `AddEventDetailsToRegistry` procedures to see how this was done. When the service is `Uninstalled` the key needs to be deleted. This is done in `RemoveEventDetailsFromRegistry`.

Creating The Event

Listing 5 shows the sequence of three calls needed to write to the event log. The `RegisterEventSource` API accepts two parameters. The first is the UNC name of the host computer (`nil` defaults to the local machine). The second is the source name mentioned earlier. This function returns a handle that needs to be passed to the following calls. The `ReportEvent` API generates the event record. The declaration for this function is shown in Listing 6.

Table 4 shows the meaning of each parameter. After generating the event you need to call `DeRegisterEventSource` to close the handle. Once the event has been generated it should be present in the event viewer under the application log. In the `DemoSv1` service I always write the `FServiceStatus` record to the event log as 'raw data' just to show how it is done.

Running DemoSv1

If you wish to run `DemoSv1` first compile it in Delphi as a console application (but do not run it from Delphi). Then execute the program in a console window and pass `INSTALL` as a parameter. A message should indicate that it was installed OK. Now, go into `Settings/Control Panel/Services`. There should be a service called Demonstration Service 1 listed with a start-up type of manual. Its current status should be blank.

To start the service select it from the list then click the `Start` button (you can pass a delay period between 500 and 10000 in the `Start-up Parameters` field). After a few seconds the SCM should show that its status is now started and it should start beeping. If you look at the event viewer an information event should have been created to show that the service was started. You should now be able to pause and then restart the service using the `Pause` and `Continue` buttons. If you restart the service you should see a message in the event viewer showing the number of milliseconds which elapsed while it was paused. After clicking `Stop` to stop the service the status should return to blank. To uninstall the service, execute it in a console window and pass `UNINSTALL` as a parameter.

That just about covers the basics as far as services are concerned. I'll now briefly describe the classes I have developed to hide these details and describe the extended examples I have provided to demonstrate some of the features described in this article.

Encapsulating Services In Delphi Classes

My aim was, as far as possible, to hide all the interface requirements to the SCM inside Delphi classes. The main problem areas are associated with lengthy operations such as start-up and close down. I wanted to be able to simply execute code (such as `DoServiceStartup`) without having to worry about co-ordinating updates to the SCM if this was going to take a long time and may be in danger of timing out. If required, an additional thread is created with the responsibility of updating the SCM with the current state of play. Virtual functions are used to indicate if a thread should be created to perform this function. The default `TNTService` class always returns `False`. If an overridden method returns true a thread is automatically created to update the SCM.

I have created two base classes to do this, `TNTServiceController` and `TNTService`. Listing 7 shows the type declarations for these two classes: see SVCClass.Pas on the disk for the implementation details.

The Services.Pas unit contains classes derived from the base classes to implement the functionality for the services. Listing 8 shows the `DemoSv2` project file to show how a typical service program would be implemented using these classes.

The first problem you have when attempting to implement Windows call-back functions into classes is the object model used by Delphi: methods have an extra hidden instance pointer (`Self`) so they can't normally be passed as call-back functions. I covered this in Issue 18 in an article called *Generic MakeMethodInstance for 16/32 bit applications*. We can use that technique here. In case you don't have that issue I have included the file MakeMic.Pas from that article on the disk.

TNTServiceController Class

The main functions of this class are as follows.

It keeps a list of services that it controls. These are added to the list via the `RegisterService` method when the program starts. `RegisterService` accepts a class reference as its parameter. This class reference needs to be for a class derived from `TNTService`. Keeping a list of class references in the object ensures that the class is able to defer the start-up of services until requested by the SCM. This keeps the number of threads active to the absolute minimum (number of started services plus 1).

It provides a private `ServiceMain` method (declared with the `stdcall` directive) and automatically makes it callable from Windows by using `MakeMethodInstance`.

It provides `InstallServices` and `UninstallServices` methods to allow for block or individual install and uninstall calls for the registered services. Configuration options can be supplied by the particular implementation of the `TNTService` derived class (`TNTService` provides default values). Dependency information can be specified for a service.

The `Connect` method connects to the SCM and starts the dispatcher. It supplies the names of all registered services each having the same `ServiceMain` entry point (the address which is returned from `MakeMethodInstance`).

When the dispatcher need to create a service it creates a thread then calls `ServiceMain`. `ServiceMain` is responsible for starting the requested service via a call to the private method `StartService` (which creates a new thread). After doing this, `ServiceMain` ends and the thread created by the dispatcher terminates. I originally wondered if this was valid but eventually found an article in the MSDN that confirms that is OK to do this.

The `StartService` method checks the name supplied against the names of the registered services. If a match is found it creates an instance of the `TNTService` derived class (ie a new thread) which becomes the main worker thread for that service. The mechanism for doing this is similar to the way `CreateForm` works (see Forms.Pas).

`DemoSv2` uses a class derived from `TNTServiceController` called `TNTServiceControllerDemo` which adds data and synchronisation objects that are shared between the services 2b and 2c.

TNTService Class

The `TNTService` class is derived from the `TThread` class. The main functions of this class are as follows.

It has a `ProcessParms` virtual method that is called at the end of the constructor to allow the service to process any parameters entered on the service start-up window. The parameters are built in the `ServiceMain` method.

It sets the `FreeOnTerminate` flag to ensure that the `TThread` object is freed when the service ends.

It provides a private `ControlHandler` method and automatically makes it callable from Windows by using a `MakeMethodInstance`. This method accepts the control requests from the SCM for this service. This entry point is unique for each service.

If required, when the `Handler` is requested to change the running status (eg Pause or Continue) and this may take a long time, it creates a second thread to keep the SCM updated with the current status.

It defines empty virtual methods that are called in response to requests from the SCM. Descendant classes should override these if required.

It provides a `LogEvent` method to allow the service to easily generate events.

It also has a class function called `ServiceName` that returns the service name. This is used in the `TNTServiceController` class when it needs to reference the service.

The class provides configuration and option information to the `TNTServiceController` via class functions and virtual methods.

Service	Description	Start-up type	Can be paused	Depends on	Uses params
2a	Beeper from DemoSv1	On demand	Yes	–	Yes (elapse ms)
2b	Monitors directory updates	Auto	Yes	–	Yes (wild card)
2c	Query service using pipes	Auto	No	2b	No

➤ *Table 5: DemoSv2 services*

```
TNTServiceController = class
  private
    FAvailableServices: TList;
    FServiceMainInstance: Pointer;
    function  ProcessOption: DWORD;
    procedure ServiceMain(NumArgs: DWord; Args: PCharArray); StdCall;
    procedure StartService(Name: Shortstring; Parms: TStrings);
  public
    constructor Create; virtual;
    destructor Destroy; override;
    procedure Connect;
    procedure InstallServices(Names: TStrings);
    procedure RegisterService(SvcClass: TNTServiceClass);
    procedure UnInstallServices(Names: TStrings);
  end;
  TNTService = class(TThread)
  private
    FController: TNTServiceController;
    FHandlerInstance: Pointer;
    FServiceStatus: TServiceStatus;
    FServicStatusHandle: SERVICE_STATUS_HANDLE;
    procedure DoTerminate; override;
    function NeedExtnededElapseTime(Option: DWORD): Boolean; virtual;
    function GetPaused: Boolean;
    procedure SetCurrentState(Value: DWORD);
    procedure StartNotificationThread;
    procedure TerminateNotificationThread;
  protected
    function AcceptPause: Boolean; virtual;
    function AcceptStop: Boolean; virtual;
    function CanInteract: Boolean; virtual;
    procedure DoHandlerNotification; virtual;
    procedure DoServiceStartup; virtual;
    procedure DoServiceProcessing; virtual; abstract;
    procedure DoServiceCloseDown; virtual;
    procedure Execute; override;
    procedure Handler(Code: Integer); stdcall;
    procedure LogEvent(Severity: DWord; Id: DWord; Inserts: PCharArray;
      NumInserts: Integer);
    procedure ProcessParms(Parms: TStrings); virtual;
    function WantShutdownNotification: Boolean; virtual;
    property CurrentState: DWORD read FServiceStatus.dwCurrentState
      write SetCurrentState;
    property Paused: Boolean read GetPaused;
  public
    constructor Create(Parms: TStrings; Controller: TNTServiceController);
      virtual;
    destructor Destroy; override;
    class procedure DependentServices(List: TStrings); virtual;
    class function ServiceDisplayName: Shortstring; virtual; abstract;
    class function ServiceName: Shortstring; virtual; abstract;
    class function ServiceStartType: DWORD; virtual;
    property Controller: TNTServiceController read FController;
  end;
```

➤ *Listing 7*

Class functions are used whenever that value may be required when there may not be an instance of the object (eg `ServiceName`).

The `Execute` method calls the `DoServiceStartUp` method then calls `DoServiceProcessing`. Descendant classes must put their main service processing in this method, not the execute method.

The class defines three abstract methods: `DoServiceProcessing`, `ServiceDisplayName` and `ServiceName` which must be overridden in descendant classes. A unique class needs to be defined for each service. `DemoSv2` defines three classes derived from `TNTService` to implement the different services which are described in Table 5.

Running DemoSv2

This program can be compiled and run in exactly the same way as `DemoSv1`. There are slight differences: After `INSTALL` or `UNINSTALL` on the command line you can supply a list of individual services to be installed or uninstalled.

After installing the services you should see three new services in the Services applet. These are listed as Demonstration Service 2a, 2b and 2c. There are slight differences in the configuration for each these services to show how the various options are implemented for each service. I have implemented a dependency between the services 2b and 2c. Also, service 2b accepts a parameter entered by the user in the Services control panel applet. Table 5 lists the specifics for each service.

Service 2b is set to monitor updates to directory C:\TEMPX (it creates the directory if it doesn't exist). If an update is made to this directory it copies all files with the archive attribute to C:TEMPX\SVBACKUP, then clears the archive attribute.The optional parameter can be a wild card (defaults to `*.*`) used to limit the files which are copied. As well as doing this it updates a log of information in a `TStrings` object. This object is stored in the `TNTServiceControllerDemo` object. Service 2c waits for requests from a client application (DemoCl2.DPR on the disk) to a named pipe. `DemoCl2` is another console application. Pass `Query` as the command line parameter to list the log details, and pass `Reset` as the parameter to clear the log details.

Due to the dependency between services 2b and 2c the SCM will ensure that the services start in the correct order. Also, if you request to stop service 2b it will prompt you to confirm that you also want to close service 2c. If you click OK the SCM will stop both services. Note: if you install service 2c and not 2b the installation will work OK but you will not be able to start service 2c. This is because the SCM cannot find the details for service 2b in the registry.

Further Information

I have supplied a file on the disk called SERVICES.TXT that lists

some articles in the MSDN, which provide additional information.

One Final Tip...

I was caught out for a while when Delphi was failing to compile a service program saying that it was unable to create the output file. It turned out that the event viewer had a lock on the file.

As the resources for the message are in the EXE files the event viewer holds the file open when it needs to display messages linked to that service.

John Chaytor is a freelance programmer who lives and works in Brighton, UK, and can be contacted via CompuServe as 100265,3642

```
program DemoSv2;
Uses SysUtils, Classes, Windows, SvcClass, Services, Logging;
var
  I: Integer;
  Option: ShortString;
  ServiceController: TNTServiceController;
  ServiceList: TStrings;
begin
  ServiceController := TNTServiceControllerDemo.Create;
  try
    Option := UpperCase(ParamStr(1));
    ServiceList := TStringList.Create;
    For I := 2 to ParamCount do
      ServiceList.Add(UpperCase(ParamStr(I)));
    With ServiceController do
      try
        RegisterService(TService2a);
        RegisterService(TService2b);
        RegisterService(TService2c);
        if Option = '' then
          Connect
        else
          if (Option = 'INSTALL') or (Option = 'I') then
            InstallServices(ServiceList)
          else
            if (Option = 'UNINSTALL') or (Option = 'U') then
              UninstallServices(ServiceList)
            else
              if (Option = 'VERSION') or (Option = 'V') then
                DisplayVersionDetails
              else
                DisplaySyntaxOptions;
      finally
        ServiceList.Free;
      end;
  finally
    ServiceController.Free;
  end;
end.
```

➤ *Listing 8*