Managing and Writing Windows services with Delphi

Michaël Van Canneyt

March 16, 2014

1 Introduction

Services are special applications which run in the background and which don't usually interact with the user. Most often, they are started when the computer is switched on, and stop when the computer is switched off. Drivers for peripherals can also operate as services, providing access to these peripherals for the operating system for for user programs. Some services can also be started and stopped manually by the user, for instance to perform periodic tasks. The Apache web-server is a service. Core components of Windows NT, 2000 or higher are also services: Indeed, the server capabilities of Windows 2000 Server are implemented as a service.

Services are managed by the Service Control Manager (SCM): It takes care of starting, stopping and generally managing services on the Windows platform; Part of it's functionality is exposed through the 'Services' applet in the control panel: It offers a limited interface to the service control manager.

Microsoft has provided a rich API to interact with the Service Control Manager: services themselves have to use this API to report to the SCM, but it can also be used to manage all services; the Control Panel applet is a simple frontend to the provided API. The definitions of the services API are in the unit WinSvc, provided standard in the VCL.

In the first sections of this article, the part of the services API used to manage services will be discussed. Since this API is oriented towards C programmers, and not very convenient for use in Delphi, a component that wraps the API in an easy-to-use class will be developed, and demonstrated in a Services Control program, similar to the one provided in Windows itself. The techniques demonstrated here can be used when writing an application to control a self-written service.

In the second part of the article, the calls in the services API that are needed for implementing a service are discussed: Two services will be discussed. The first service is very simple, using a straightforward pascal approach, to demonstrate how a service application works: A service which accepts client connections on a named pipe and gives them the current time. The second service is written using the components provided in Delphi: Borland has created a wrapper around the services API which makes writing services in Delphi quite easy. A service will be written which periodically cleans the harddisks by zipping files it finds below certain directories and which match certain criteria (size, extension). A control program to manage the service's configuration is also provided.

2 The `TServiceManager` component

The component `TServiceManager` can be dropped on a form to implement a service controller, or simply to provide a simple means of installing a service: it supports many of the service API calls and wraps them in more Object Pascal-like interface: Instead of null-terminated strings, normal ansistrings are used, in case of an error, exceptions are raised.

To be able to interact with the SCM, a connection handle must be obtained via the Windows `OpenSCManager` call. This handle must be used in all calls to the SCM. The `TServiceManager` component publishes a boolean 'Connected' property: Setting it to `True` will connect to the SCM. The obtained handle is then available through the (public) `Handle` property. If needed, it can be used to implement extra calls not implemented in the `TServiceManager` component itself. Setting the `Connected` property to `False` will disconnect from the SCM.

To be able to control services, one must know which services are installed. For this, the `TServiceManager` has a wrapper around the windows `EnumServicesStatus` call: this call returns a list of all installed services, and their current status. The call is defined as follows:

```
function EnumServicesStatus(hSCManager: SC_HANDLE; dwServiceType,
  dwServiceState: DWORD; var lpServices: TEnumServiceStatus; cbBufSize: DWORD;
  var pcbBytesNeeded, lpServicesReturned, lpResumeHandle: DWORD): BOOL; stdcall;
```

This call has a lot of arguments, the most important being the `lpServices` argument: When the function returns successfully, this variable will point to an array of `TEnumServiceStatus` records, which contain the definitions and states of the installed services.

The `TServiceManager` component wraps this functionality in a 'Refresh' method: This method will collect the information returned by this call, and store it in the `Services` property: The `Services` property is a descendent of `TCollection` which manages a series of `TServiceEntry` collection items. The `TServiceEntry` contains the information about a service, and is defined as follows:

```
TServiceEntry = Class(TCollectionItem)
Public
  Property ServiceName : String;
  Property DisplayName : String;
  Property ServiceType : DWord;
  Property CurrentState : DWord;
  Property ControlsAccepted : DWord;
  Property Win32ExitCode : DWord;
  Property ServiceSpecificExitCode;
  Property CheckPoint : DWord;
  Property WaitHint: DWORD;
end;
```

For each installed service, the `TServices` collection will have an item. The main properties of this collection item are:

ServiceName an internal (unique) name for the service.

DisplayName The name of the service as it is displayed on screen.

ServiceType an integer describing the type of service. There are 4 possible values for this property:

- SERVICE_FILE_SYSTEM_DRIVER for filesystem drivers.
- SERVICE_KERNEL_DRIVER for other kernel drivers.
- SERVICE_WIN32_OWN_PROCESS for an executable which offers a single service.
- SERVICE_WIN32_SHARE_PROCESS for an executable which offers several services (more about this later)

In this article, only the latter two types will be discussed.

CurrentState describes the current state of the service. This can be one of the following 7 values:

- SERVICE_STOPPED when the service is not running.
- SERVICE_START_PENDING when the service is starting up.
- SERVICE_STOP_PENDING: The service is in the process of shutting down.
- SERVICE_RUNNING: The service is running normally.
- SERVICE_CONTINUE_PENDING: The service is about to resume its operations.
- SERVICE_PAUSE_PENDING: The service is about to suspend its operations.
- SERVICE_PAUSED: The service has suspended it operations, but is still running and in memory.

ControlsAccepted This is a bitmask of various control commands to which the service will respond. The following values can be included:

- SERVICE_ACCEPT_STOP The service can be stopped.
- SERVICE_ACCEPT_PAUSE_CONTINUE The service can be paused and/or continued.
- SERVICE_ACCEPT_SHUTDOWN the service can be shut down.

The other properties are less important, for more information, the Windows API reference contains the complete explaination of all available fields.

The Services property is of type TServiceEntries, a TCollection descendent, which additionally implements the following interface:

```
TServiceEntries = Class(TOwnedCollection)
Public
  Function FindService(ServiceName : String) : TServiceEntry;
  Function ServiceByName(ServiceName : String) : TServiceEntry;
  Property Items [index : Integer] : TServiceEntry;default;
end;
```

Note the default property. The Refresh method of the TServiceManager component will clear the list of service entries, and refill it with current information. The list can be cleared using the ClearServices method.

To refresh the status of a single service in the collection, the RefreshServiceStatus method is implemented in TServiceManager:

```
RefreshServiceStatus(ServiceName: String);
```

The servicename argument should match the property of the same name in TServiceEntry. It invokes the QueryServiceStatus call of the services API to get the needed status information.

3 Service definitions

With the above properties, a list of installed services and their current status can be obtained. It is also possible to get more detailed information about a service: Windows provides the `QueryServiceConfig` call to obtain the configuration details of any installed service; It is defined as follows:

```
function QueryServiceConfig(hService: SC_HANDLE;
  lpServiceConfig: PQueryServiceConfig; cbBufSize: DWORD;
  var pcbBytesNeeded: DWORD): BOOL; stdcall;
```

The `lpServiceConfig` must point to a buffer in memory, large enough to receive all configuration information about the service. The initial part of this memory block will be a structure of type `TQueryServiceConfig`. To make this call a bit more pascal-like, the `QueryServiceConfig` call was wrapped in a method of the same name. It is defined as follows:

```
Procedure QueryServiceConfig(ServiceName : String;
                             Var Config : TServiceDescriptor);
```

Upon return, the `TServiceDescriptor` record - defined in the ServiceManager unit - will be filled with all configuration data for the service. It is defined as follows:

```
TServiceDescriptor = Record
  Name : ShortString;
  DisplayName : ShortString;
  DesiredAccess : DWord;
  ServiceType : DWord;
  StartType : DWord;
  ErrorControl : DWord;
  CommandLine : String;
  LoadOrderGroup : String;
  TagID : DWord;
  Dependencies : String;
  UserName : String;
  Password : String;
end;
```

The meaning of the various fields partially overlaps with the various properties of the same name in the `TServiceEntry` item. The meaning of the other fields is as follows:

DesiredAccess The desired access rights to the service. This field is not used by the `QueryServiceConfig` call, but is used by the `RegisterService` call, described below.

StartType The startup type of the service. This can be one of the following values:

- `SERVICE_AUTO_START` The service will be started automatically by the service controller at boot.
- `SERVICE_BOOT_START` The service will be started by the system loader. This should only be used for drivers.
- `SERVICE_DEMAND_START` The service should be started manually.
- `SERVICE_SYSTEM_START` The service is started by the driver manager (only for drivers).

- SERVICE_DISABLED The service is temporarily disabled, any attempts to start will result in a failure.

ErrorControl Indicates what action should be taken by the service controller when the service fails to start. It can have one of the following values:

- SERVICE_ERROR_IGNORE The error is logged, but ignored.
- SERVICE_ERROR_NORMAL The error is logged, and a message is shown on the screen, but the system continues to boot.
- SERVICE_ERROR_SEVERE The error is logged, but the boot process will fail, unless the boot was started in safe mode.
- SERVICE_ERROR_CRITICAL The error is logged, and the boot process will fail, even in safe mode.

CommandLine The command-line used to start the binary that contains the service. If the path to the binary contains spaces, it should be enclosed in double quotes ("). Options may be appended to the command-line.

LoadOrderGroup The load order group. When the service is started at boot, it will be started as part of the name group. The list of available groups can be found in the registry, under HKEY_LOCAL_MACHINE entries under key:

```
\System\CurrentControlSet\Control\ServiceGroupOrder
```

The binary value named List contains a null-separated list of strings that list the names of load groups.

TagID Here a numerical ID is returned that identifies the order of loading within the load group.

Dependencies A list of slash-separated (/) names of services or service groups that must be running before the service can be started. A service group has a '+' character prepended to its name.

UserName The username the service should be started as. This must be a username of the form

```
Domain\UserName
```

The domainname may be a signle dot (.) to indicate the local machine. If the UserName is empty, the LocalSystem account is used.

Password The password to be used for the username. For the LocalSystem account, the password should be empty

The same structure can be used to register a new service in the system. To register a new service, Windows implements the CreateService call. It is defined as follows:

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

The arguments for this call correspond to the various fields in the TServiceDescriptor record. The CreateService call has been wrapped in the RegisterService method of TServiceManager:

```
Function RegisterService(Var Desc: TServiceDescriptor) : THandle;
```

If succesful, the function returns a handle to the newly created service. The handle will be created with the access rights specified in the `DesiredAccess` field. More information about access rights is given below.

The service may be reconfigured at any later time using the `ConfigService` method, which is much like the `RegisterService` method:

```
Procedure ConfigService(ServiceName : string; Config: TServiceDescriptor);
```

Calling this service will change the configuration of the service `ServiceName` with the values supplied in the `Config` record. To leave a string value unchanged, specify an empty string. To leave a `DWORD` value unchanged, the predefined constant `SERVICE_NO_CHANGE` must be supplied.

Obviously, a registered service can also be unregistered. For this, the `UnregisterService` call is implemented:

```
Procedure UnregisterService(ServiceName : String);
```

This will unregister the service and delete all configuration data from the list of installed services. If the service is still running and cannot be stopped, then the data will be marked for deletion, and the service will be deleted at the next boot of Windows.

A common task is to change the startup-type of a service. This could be done with the `ConfigService` call, but this is rather cumbersome. Therefore, the startup type of a service can be set in a more convenient way with the `SetStartUpType` method, defined as follows:

```
procedure SetStartupType(ServiceName: String; StartupType: DWord);
```

4 Controlling services

The methods presented in the previous section show or change the configuration information of a service. Once a service is running, it can also be controlled: It can be stopped, shut down or paused and restarted. The windows API call to control a service is defined as follows:

```
function ControlService(hService: SC_HANDLE;
                        dwControl: DWORD;
                        var lpServiceStatus: TServiceStatus): BOOL;
```

When the `ControlService` returns, the last reported state of the service is reported in the buffer pointed to by `lpServiceStatus`. The `dwControl` parameter specifies the kind of action that should be performed on the service. It can be one of the following values:

SERVICE_CONTROL_STOP The service should stop its activities and exit.

SERVICE_CONTROL_PAUSE The service should cease its activities, but remain standby (pause).

SERVICE_CONTROL_CONTINUE Tells a previously paused service that it can resume its activities.

SERVICE_CONTROL_INTERROGATE Tells the service that it should report its status as soon as possible.

The `ControlsAccepted` property of `TServiceEntry` contains a bitmask indicating which control codes can be sent to the service. It is also possible to send service-specific control codes. They must have a value in the range `[128..255]`. The meaning of these codes depend on the implementation of the service: They can be sent to a running service to perform certain actions immediatly, or to notify the service that some configuration has changed. The `TServiceManager` component implements a call for this:

```
procedure CustomControlService(ServiceName : String; ControlCode : DWord);
```

The call will check whether the controlcode is in the allowed range before ending it to the service.

The pre-defined control codes are wrapped in a series of calls:

```
procedure ContinueService(ServiceName : String);
procedure StopService(ServiceName: String; StopDependent: Boolean);
procedure PauseService(ServiceName: String);
```

The `StopService` call accepts an additional argument: `StopDependent`, a boolean indicating whether services that depend on the service to be stopped, should be stopped first. The method will attempt to stop these services first.

The list of services that depend on a given service can be obtained using the `ListDependentServices` method, given as

```
Procedure ListDependentServices(ServiceName : String;
                                ServiceState : DWord;
                                List : TStrings);
```

The list of dependent services can be filtered on their state by specifying a state bitmask in the `ServiceState` argument; Allowed values are the same ones as in the `CurrentState` property of `TServiceEntry`.

Starting a stopped service is not done using the `ControlService` call. There is a separate call for this, defined as follows:

```
procedure StartService(ServiceName : String; Args: TStrings);
```

The `StartService` call accepts list of arguments, which will be passed on to the service. This list may be empty (A value of `Nil` for `Args`).

It is important to note that the list of arguments given here is *not* the same as the options passed on to the binary containing the service - as specified in the `CommandLine` field in the `TServiceDescriptor` record. More on this will be said later in the section on writing a service.

5 `TServiceManager` odds and ends

The methods and structures presented in the previous sections constitute the bulk of the `TServiceManager` class. There are some extra calls implemented, which are less often needed. The first pair of these calls can be used when changing configuration information for a group of services. In order to prevent services from being started or stopped during this period, the service manager can be told to lock the services database. There are 2 calls for this:

```
Procedure LockServiceDatabase;
Procedure UnlockServiceDatabase;
```

In order to prevent lock-up of the database, it is best to put the `UnlockServiceDatabase` in the `Finally` part of a `Try...Finally` construction.

Most operations on a service require a handle to the service. A handle can be obtained using the `GetServiceHandle` method:

```
function GetServiceHandle(ServiceName: String; SAccess: DWord): THandle;
```

The methods of the `TServiceManager` class obtain and free handles as needed. Each of the methods that has a `ServiceName` argument has also an overloaded variant which accepts a handle to the service instead: If a lot of operations must be done on a service, a handle can be obtained manually, and passed on to the various overloaded calls.

The `GetServiceHandle` call has an `SAccess` argument: it is used to specify the desired access rights to the service: Each operation has its own access right, and the correct access rights must have been requested for an operation to succeed. Rights can be combined: they are a set of bitmasks. Discussing the various possible access rights would lead too far; the Microsoft API documentation states the needed rights for each call, so they will not be documented here. The various methods of the `TServiceManager` always request exactly enough rights to be able to perform their task. When obtaining a handle to do a group of operations on a service, enough rights should be requested to perform all operations.

Indeed, to access the service manager itself, certain access rights are needed. The needed rights can be set in the `Access` property of the `TServiceManager` component. It should be set before connecting to the service controller. If the service controller to which a connection should be made resides on a remoe machine (this is possible) then the name of that machine may be indicated in the `MachineName` property of `TServiceManager`. The list of services can be automatically refreshed after connecting if the `RefreshOnConnect` property is set to true.

Finally, three events have been implemented to react on changes in the list of services:

```
Property AfterRefresh : TNotifyEvent;
Property AfterConnect : TNotifyEvent;
Property BeforeDisConnect : TNotifyEvent;
```

Their meaning should be obvious.

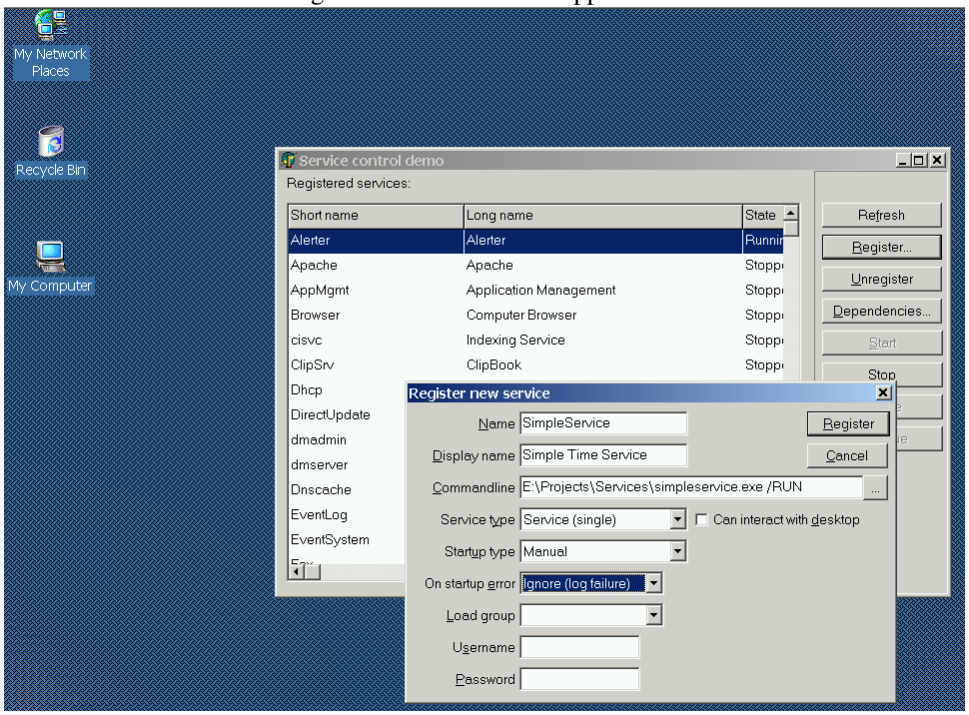
Most of the methods described in the above sections are demonstrated in a small application - a Delphi implementation of the Control Panel 'services' applet. It has slightly more functionality: A new service can be registered as well; This will come in handy for the service written in the next section. The application can be seen running in figure 1 on page 9.

6 A simple service

Writing a service is not so different from writing a normal application, with the proviso that there are a number of things that must be done:

1. The implemented service (or services) must be registered with the SCM. This means registering a service entry point: A procedure which will be used by the control manager to start the service. This should be done at progam startup.

Figure 1: Service control application



2. Each service must register a control entry point. This is a procedure which will be called by the control manager to send control commands (pause,continue,stop) to the service. This should be done as soon as the service starts.

3. At regular intervals in the service start procedure (or on demand) the service must report its state to the SCM. It is important that this be done as soon as possible, since the SCM will decide that the service failed to start properly if it doesn't receive status information after a reasonable time.

After these tasks are completed, the service can perform it's action in the service entry point: This is the function which will do the actual work, and when it returns, the service is stopped.

To illustrate this, a simple service will be implemented: A service that opens a named pipe, and listens on the pipe for client connections. As soon as a client program connects, the current time is written to the pipe, and the connection is closed.

To write this program, a new 'console application' is started, which will open the project file. The `{$APPTYPE CONSOLE}` directive which Delphi automatically inserts in the must be removed, or a console will be opened each time the service is run.

The main pogram block is very simple:

```
begin
  RunAsService:=(ParamCount=1) and (Paramstr(1)='/RUN');
  If RunAsService then
    RegisterServiceLoop
  else
    SimpleServiceMain(0,Nil);
end.
```

To run the program as a service, the `/RUN` command-line parameter must be given: this parameter should be specified when the service is registered. If the parameter is not specified, then the service's main function is executed by the program itself: this wil run the application as a normal application. If the `/RUN` parameter is given, then the service main function is registered with the SCM in the `RegisterServiceLoop` procedure. This is a simple procedure that looks as follows:

```
Var
  ServiceTable : Array[0..1] of TServiceTableEntry = (
   (lpServiceName: 'SimpleService';lpServiceProc:@SimpleServiceMain),
   (lpServiceName: Nil;lpServiceProc:Nil)
   );

Procedure RegisterServiceLoop;

begin
  If not StartServiceCtrlDispatcher(ServiceTable[0]) then
    ReportOSError(SErrRegistering);
end;
```

The `StartServiceCtrlDispatcher` call takes as an argument an array of `TServiceTableEntry` records, ended by a dummy record filled with `Nil` value. For each service provided by the application, there must be a record of type `TServiceTableEntry` which looks as follows:

```
TServiceTableEntry = record
  lpServiceName: PAnsiChar;
  lpServiceProc: TServiceMainFunctionA;
end;
```

The first field is the name of the service, and the second field is the service entry point (`SimpleServiceMain` for the simple service).

When the service must be started, the SCM will call the service entry point. The simple service application calls this procedure itself when it is not run as a service). The service entry point must be of type `TServiceMainFunctionA`, which means it must accept 2 arguments:

argc A `DWORD` that will contain the number of arguments to the function, i.e. the number of entries in the second argument.

Argsv A `PPChar` pointer that points to a null-terminated array of null-terminated strings. There is always at least 1 null-terminated string in the list: the first entry contains the name of the service. This can be used to provide a single entry point for several services: from the first argument, the name of the service to be started can be determined.

These parameters are similar to the command-line arguments passed to a program when it starts. However, they are *not* the same as the command-line arguments passed to to the application when it is initially run. The arguments passed to the service entry point are given by the `StartService` call of the SCM.

Since it will be called by the SCM, care must be taken that the calling convention of the service entry point is `StdCall`.

For the simple service, the service entry point looks as follows:

```
Procedure SimpleServiceMain (Argc : DWord; Argsv :PPChar);stdcall;

Var
  BytesTransferred : DWord;
  Command : Dword;

begin
  ControlPort:=CreateIoCompletionPort(INVALID_HANDLE_VALUE, 0, cmdPipe, 0);
  If (ControlPort=0) then
    ReportOSError(SErrControlPort);
  If RunAsService then
    begin
    // Initialize status record.
    FillChar(CurrentStatus,SizeOf(CurrentStatus),0);
    With CurrentStatus do
      begin
      dwServiceType:=SERVICE_WIN32_OWN_PROCESS;
      dwCurrentState:=SERVICE_START_PENDING;
      dwControlsAccepted:=SERVICE_ACCEPT_STOP or
                          SERVICE_ACCEPT_PAUSE_CONTINUE;
      end;
    ServiceStatusHandle:=RegisterServiceCtrlHandler('SimpleService',
                                                    @SimpleServiceCtrlHandler);
    SetServiceStatus(ServiceStatusHandle,CurrentStatus);
    end;
  CreatePipe;
  If RunAsService then
    begin
    CurrentStatus.dwCurrentState:=SERVICE_RUNNING;
    SetServiceStatus(ServiceStatusHandle,CurrentStatus);
    end;
  PO:=Nil;
  Repeat
    // Wait for either control code notification or client connect
    GetQueuedCompletionStatus(ControlPort,BytesTransferred,Command,po,INFINITE)
    If Command=cmdPipe then
      HandlePipe;
     // otherwise a Control code received, do nothing, wait for new command
  Until RunAsService and (CurrentStatus.dwCurrentState=SERVICE_STOPPED);
end;
```

The function starts out by creating a control port; More about this follows. If the program is run as a service, the first thing that is done is reporting the service status to the SCM. This is done by filling a `TServiceStatus` record with appropriat values:

1. The service type must be passed.
2. The current state: `SERVICE_START_PENDING`.
3. The control codes that will be accepted by the services. For the simple service, the stop, pause and continue control commands are accepted.

The record will then be passed to the SCM with the `SetServiceStatus` call. To be able to send it to the SCM, a handle to the service must be obtained. This is done by registering the control entry point: the `RegisterServiceCtrlHandler` call registers the control

entry point for the service (it is called SimpleServiceCtrlHandler). It will return a handle to the service, which can be used in the `SetServiceStatus` call.

After this is done, the named pipe is created on which clients can connect to the service. If the application is running as a service, the status will subsequently be set to running, and reported to the SCM.

After this is done, the actual work of the application is started. It enters a loop, in which it waits for clients to connect to the named pipe. It does does this using a IOCompletionPort: This is a special port which an application can request from windows, and by which windows can notify the application that a certain IO operation is completed. In our case, the operation is a client which makes a connection on the named pipe. The port was created with the `CreateIoCompletionPort` at the start of the service entry. The `cmdPipe` constant used in that call tells windows that when someone connects to the pipe, the `GetQueuedCompletionStatus` call should receive the `cmdPipe` value in the `Command` argument:

```
GetQueuedCompletionStatus(ControlPort,BytesTransferred,Command,po,INFINITE);
If Command=cmdPipe then
  HandlePipe;
```

The `INFINITE` argument tells Windows that the `GetQueuedCompletionStatus` call should wait forever before returning. When the call returns, the value stored in `Command` is checked: if it is `cmdPipe`, a client has connected to the named pipe, and the connection to the pipe is handled in the `HandlePipe` call.

The loop is run until the service is stopped, or forever if the application is run as a normal application. More information about the `CreateIoCompletionPort` and `GetQueuedCompletionStatus` will not be given here, as that would lead too far. The interested reader can consult the Microsoft API documentation for more information.

When the service needs to be stopped or paused, the SCM will send a control code to the control entry point that was registered by the service. The service control entry procedure should accept a single argument: A `DWord` value, which is the control code being sent by the SCM. As the control entry point is called by the SCM, it should have `stdcall` calling conventions. The control entry point should do whatever is necessary for the received control code, and report the service status as soon as possible with the SCM. Note that since there is only one argument to the control entry point, there is no way to know for which service the control code is sent, in case multiple services are provided by the application: each service must have its own control entry point.

The control entry point for the simple service looks as follows:

```
Procedure SimpleServiceCtrlHandler (Opcode : DWord); StdCall;

begin
  Case Opcode of
    SERVICE_CONTROL_PAUSE :
      begin
      ClosePipe;
      CurrentStatus.dwCurrentState:=SERVICE_PAUSED;
      end;
    SERVICE_CONTROL_STOP :
      begin
      ClosePipe;
      CurrentStatus.dwCurrentState:=SERVICE_STOPPED;
      end;
```

```
    SERVICE_CONTROL_CONTINUE :
      begin
      CreatePipe;
      CurrentStatus.dwCurrentState:=SERVICE_RUNNING;
      end;
    SERVICE_CONTROL_INTERROGATE : ;
  else
    ServiceError(SErrUnknownCode,[Opcode]);
  end;
  SetServiceStatus(ServiceStatusHandle,CurrentStatus);
  // Notify main thread that control code came in, so it can take action.
  PostQueuedCompletionStatus(ControlPort,0,cmdControl,NiL);
end;
```

As can be seen, it is a simple routine: Depending on the control code received, the pipe is closed or created, and the new status of the service is stored in the global `CurrentStatus` record, after which the status is reported to the SCM. Then, the main loop is notified that there was a change in status: the `PostQueuedCompletionStatus` will queue a IO completion event to the IOCompletionPort, with code `cmdControl`. This will cause the `GetQueuedCompletionStatus` call in the main program loop to return. If the service is being stopped, the main loop will then exit, and the service will stop. In any other case, the main loop will simply repeat the `GetQueuedCompletionStatus`, waiting for a connection on the pipe or till another control code arrives.

The rest of the code in the simple service is simply concerned with the handling of the pipe: creating a pipe, closing it down or handling a client connection on the pipe. The code is quite simple, and the interested reader is referred to the source code which is available on the CD-ROM accompagnying this issue.

To test the simple service, it can be installed with the service manager program developed above, using the name 'SimpleService', and giving it the `/RUN` command-line option. The CD-ROM also contains a simpleclient client program which will connect to the service and show the time it received from the service.

7 Services in Delphi

Borland has wrapped the API calls and functionality to write services in 2 classes, shipped with Delphi's VCL. They are contained in the `SvcMgr` unit. The classes are:

TServiceApplication is a class which performs the same function as the `TApplication` provides for a normal GUI application. It has all functionality to register services, start services and respond to control codes. A global instance of this type is instantiated in the `SvcMgr` initialization code, just as is done for its GUI counterpart in the Forms unit.

TService is a `TDataModule` descendent which implements a service: it corresponds to the `TForm` class in a normal GUI application. Each service application must contain one or more descendents of the `TService` Class. It implements all functionality of a single service: the main loop, events corresponding to the control codes that can be sent by the SCM.

To create a service application, select 'New service Application' from the 'New' menu. Delphi will then create a new project which contains 1 service. Looking at the project source, something similar to the following can be seen:

```
program diskclean;

uses
  SvcMgr,
  svcClean in 'svcClean.pas' {CleanService: TService};


begin
  Application.Initialize;
  Application.Title := 'Disk cleaning service';
  Application.CreateForm(TCleanService, CleanService);
  Application.Run;
end.
```

Which looks a lot like the code for a normal GUI application, with the exception that the `Application` object lives in the `SvcMgr` unit. Choosing `New service` from the `New` menu in Delphi will add new services to this list.

The `Run` method takes care of registering the services, unregistering them, or running the services. Only services registered with the `Application.CreateForm` method will be installed or run.

A Delphi service application can install itself: supplying the `/INSTALL` command-line option will register all available services and exit after displaying a messagebox announcing that the services have been installed. The `/SILENT` option suppresses the message. Likewise, the `/UNINSTALL` command-line option will unregister the services. The service class has some handlers which can be used to respond to these events:

BeforeInstall

AfterInstall

BeforeUninstall

AfterUninstall

The meaning of these events should be clear from their name. The various other properties such as `Name`, `DisplayName`,`ServiceType` and `StartType` of the `TService` class should be set to an appropriate value, as they will be used when registering the service.

Giving other command-line options to the service application will actually run the service application: In that case the `Run` method of the `TServiceApplication` class will start a thread which registers the main entry point for all services using the `StartServiceCtrlDispatcher` call discussed in the previous section. One entry point is used for all services, and the options passed to the entry point are used to dispatch the call to the correct `TService` class so it can do its work. After all service entry points were registered an event loop is started, waiting for windows events to arrive. As soon as the thread stops, the application stops.

When the SCM starts a service, the call to the service entry point is dispatched to the `Main` method of the appropriate service instance. Which service instance to use is determined from the arguments to the entry point.

The `Main` method of the service is once more a big event loop, started in a separate thread (available in the `ServiceThread` property of the service). The loop is interspiced with some event handlers and status reporting events. The following events exist:

OnStart This method is called once when the service is started. The `Started` parameter should be set to `True` if the service can be started succesfully. Setting it to `False` will abort the service.

OnExecute This is the main method of the service: here the service should do whatever it is designed to do. It should regularly call `ServiceThread.ProcessRequests`: this will process any windows events or incoming control codes.

When the SCM sends some control commands to the service, the service thread will call one of the following event handlers:

OnPause When the pause command was received. The service should stop what it is doing, but should not exit.

OnContinue When the continue command was received.

OnShutdown When the service is being shut down.

OnStop When the service is stopped. It is the

Each of these handlers is passed a boolean variable which should be set to `True` (default), indicating that the command was correctly processed, or `False` to indicate that the command failed. The service thread will then report the status (using the `ReportStatus` method) to the SCM.

As pointed out in the previous section, the service control entry point has only one argument (the control code), making it impossible to use a single entry point for all services in the application. For this reason, the Delphi IDE inserts a control entry point for each service that is created. The control entry point is inserted at the start of the implementation section of the unit containing the service and looks similar to this:

```
procedure ServiceController(CtrlCode: DWord); stdcall;
begin
  CleanService.Controller(CtrlCode);
end;

function TCleanService.GetServiceController: TServiceController;
begin
  Result := ServiceController;
end;
```

The `GetServiceController` function is used by the `Main` function of the service to obtain the control entry point which it registers with the SCM. The service entry point simply calls the `CleanService.Controller` method: Note that this makes dynamically creating and running multiple instances of the same service class in code is not possible, as the `CleanService` variable is then not valid. Obviously, this code should not be deleted as the service will not be able to receive any control events.

8 A Disk cleaning service

To illustrate the concepts of a service application, a disk cleaning service application is developed: The service will sit in the background, and at a certain time will scan specified directories on the hard disks for files matching certain extensions, and if they exceed a certain size, it will compress them and remove the original. The time to perform the scan, as well as the minimum size and extension of the files to be processed can be configured: Global values and per-directory values can be specified. As with all services created with Delphi, it can be installed by running it once with the `/INSTALL` command-line option.

The service can be compiled using TurboPower's (freely available) Abbrevia: it will then create a zip file for each file it must process. By default it will use the `ZLIb` unit which can

be found on the Delphi installation CD: this will create a simple compressed file. Setting the `USEABBREVIA` conditional define will compile using Abbrevia.

The `TCleanService` class in the svcClean implements the service. It contains a timer (`TClean`) and event logging class `ELClean`, discussed in an earlier edition of `Toolbox`. The `TService` class itself has a logging method, but it doesn't offer the functionality of the `TEventLog` class, hence it is not used.

When the service is started, it gets the time when it should clean the disk from the registry:

```
procedure TCleanService.ServiceStart(Sender: TService;
  var Started: Boolean);
begin
  Started:=InitTimer;
  If Not Started then
    PostError(SErrFailedToInitTimer);
end;
```

The `PostError` uses the `ELClean` component to log an error message. The `InitTimer` method retrieves the time to run from the registry, and activates the timer so a timing event will arrive when it is time to clean the disk.

The `Execute` handler contains a simple loop:

```
procedure TCleanService.ServiceExecute(Sender: TService);
begin
  While not Terminated do
    ServiceThread.ProcessRequests(True);
end;
```

The `ServiceThread.ProcessRequests` call will wait for windows event messages and control commands; This ensures that when the timer event occurs, the associated `OnTimer` event handler will be executed:

```
procedure TCleanService.TCleanTimer(Sender: TObject);

begin
  If Assigned(FCleanThread) then
    PostError(SErrCleanThreadRunning);
  StartCleanThread;
  // Reset timer.
  If Not InitTimer then
    PostError(SErrFailedToInitTimer);
end;
```

The actual cleaning will be done in a separate thread: It is created in the `StartCleanThread` method. After that, the timer is re-initialised to deliver an event on the next day, same time.

The `StartCleanThread` operation simply starts a thread, and assigns some status reporting callbacks:

```
procedure TCleanService.StartCleanThread;

begin
  FCleanThread:=TCleanThread.Create(True);
  FCleanThread.OnTerminate:=Self.CleanThreadFinished;
  (FCleanThread as TCleanThread).OnErrorMessage:=PostError;
```

```
  (FCleanThread as TCleanThread).OnInfoMessage:=PostInfo;
  FCleanThread.Resume;
end;
```

The `TCleanThread` is implemented in the `thrClean` unit, and is discussed below. Note that the `OnTerminate` event of the service is assigned. It serves to set the `FCleanThread` to `Nil` when the service finished its job:

```
procedure TCleanService.CleanThreadFinished(Sender : TObject);

begin
  With FCleanThread as TCleanThread do
    PostInfo(Format(SFinishedClean,[FTotalFiles,FormatDateTime('hh:nn:ss',FTotalTime
  FCleanThread:=Nil;
end;
```

At the same time, some cleaning statistics are logged: the `FTotalTime` and `FTotalFiles` fields of the CleanThread contain the total time the thread was working and the number of cleaned files, respectively.

When the service is stopped, the `OnStop` event handler is called:

```
procedure TCleanService.ServiceStop(Sender: TService;var Stopped: Boolean);
begin
  TClean.Enabled:=False;
  If Assigned(FCleanThread) then
    With FCleanThread do
      begin
      If Suspended then
        Resume;
      Terminate;
      WaitFor;
      end;
  Stopped:=True;
end;
```

The timer is stopped, and if the clining thread happens to be running, it is terminated. After the thread stopped executing, the handler reports success and exits.

This is everything that is needed to implement our service. In order to support pause/continue commands, the `OnPause`/`OnContinue` handlers can be used:

```
procedure TCleanService.ServicePause(Sender: TService;
  var Paused: Boolean);
begin
  TClean.Enabled:=False;
  If Assigned(FCleanThread) then
    FCleanThread.Suspend;
end;

procedure TCleanService.ServiceContinue(Sender: TService;
  var Continued: Boolean);
begin
  If Assigned(FCleanThread) then
    FCleanThread.Resume
```

```
  else
    InitTimer;
end;
```

The code of the event handlers speaks for itself.

When the user changes the time on which the service should clean the disk, and the service is running, it should be notified of this, so it can reset the timer. To do this, a custom control code can be sent to the service. Strangely enough, Borland has not implemented an event to deal with custom control codes: there is no `TService` event associated with such a command. Instead, the `DoControlCode` method must be explicitly overridden. For the cleaning service, it would be implemented as follows:

```
function TCleanService.DoCustomControl(CtrlCode: DWord): Boolean;
begin
  Result:=(CtrlCode=ConfigControlCode);
  If Result and TClean.Enabled then
    begin
    InitTimer;
    PostInfo(SConfigChanged);
    end;
end;
```

The `ConfigControlCode` constant is defined in the `cfgClean` unit. As can be seen, when this code is received, the timer is simply re-initialised, and a message recording this fact is logged.

The cleaning thread (implemented in unit `thrClean` is actually a simple loop. It contains the following methods (among others):

Execute The main function of the thread. It initializes some variables, and then loops over the list of directories listed in the registry.

RecursivelyCleanDir Called for each directory. It will clean all files in the directory, and recursively call itself to handle subdirectorie.

CleanDir Called for each directory, it scans all files in the directory, and if the file matches the search criteria, it is cleaned.

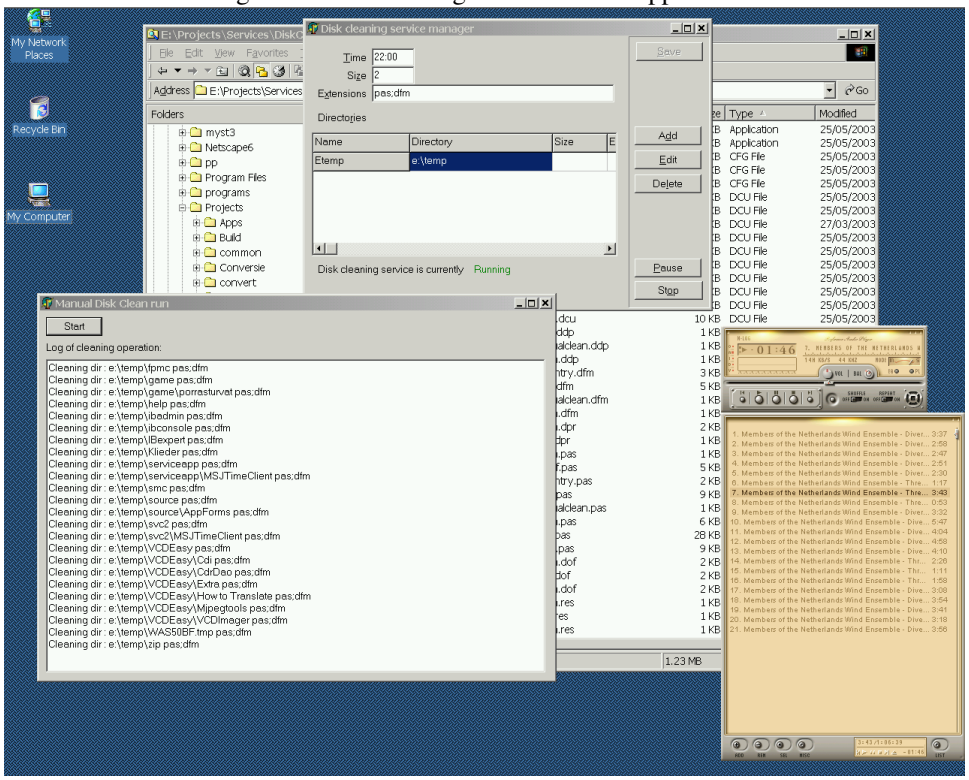
CleanFile Called for each file that matches the search criteria. It determines whether the file needs to be cleaned and cleans it if needed.

CompressFile This method actually compresses the file. Two implementations exist: one using Abbrevia, one using an internal method, based on the ZLib unit, delivered on the Delphi CD-ROM.

The interested reader is referred to the unit itself. The service can easily be adapted by changing the `CleanFile` and `CompressFile` methods. For instance, the files could be moved to a special directory instead of compressing them. They could also be archived in 1 big archive file, instead of creating a zip file per file. They could be simply deleted. It is simple code, easily adapted to suit a particular need.

The CD-ROM contains also the code for a managing application (called dkmngr): it allows to set the needed registry entries, and to start or stop the service. It uses the `TServiceManager` component introduced at the beginning of this article to do this. A screenshot of the application can be seen in figure figure 2 on page 19.

Figure 2: Disk cleaning service control application



9 Creating a dual application

There may be occasions when a dual application should be made; or one which should also run on Windows 95,98 or millenium, operating systems which do not allow for services. For instance, it would be good to be able to run the disk cleaning service manually, and see the output on the screen. The simple service presented earlier in this article also could be run as a service, or normally. This explains why the disk cleaning code was implemented separate thread instead of directly in the service object: The thread can be started from the service, or from the main form of the 'normal' application.

Implementing a dual application is not so hard, it involves some simple changes to the project source code. To demonstrate it, a form (`TManualCleanForm`, in unit `frmManualClean`) was added to the the diskclean application, which contains a simple memo and a start button. Pressing the start button will start the cleaning thread, and the progress of the cleaning progress will be shown in the memo. The form in action is also visible in figure figure 2 on page 19.

The main program code of the project file must be changed as follows:

```
begin
  Installing:=IsInstalling;
  if Installing or StartService then
    begin
    SvcMgr.Application.Initialize;
    SvcMgr.Application.Title := 'Disk cleaning service';
    svcMgr.Application.CreateForm(TCleanService, CleanService);
    svcMgr.Application.Run;
```

```
    end
  else
    begin
    Forms.Application.ShowMainForm := True;
    Forms.Application.Title:='Manual Disk cleaning service run';
    Forms.Application.Initialize;
    Forms.Application.CreateForm(TManualCleanForm, ManualCleanForm);
    Forms.Application.Run;
    end;
end.
```

The installing Flag is set from a function `IsInstalling` which scans the command-line for the `/INSTALL` or `/UNINSTALL` options. The `StartService` is discussed below: it decides whether or not the application is run normally, or as a service. If the application is run as a service, the code inserted by Delphi is used, but the application object of the SvcMgr unit is explicitly specified: The `Forms` unit was added to the `Uses` clause of the program, and this unit also contains an instance of the `Application` object.

If the `StartService` routine decided that the application is not started as a service, then the code that would be inserted in a normal GUI application is run: It looks similar to the one for a service application, only instead of the `Application` object from the SvcMgr unit, the application object of the Forms unit is used. And, of course, the main form is instantiated instead of the service.

The implementation of the `StartService` routine is not straighforward, and somewhat of a kludge: Unfortunately, Borland's implementation of the service registration routine does not allow to specify command-line parameters when registering the service. It uses simply the binary name, without command-line options. The simple service program presented earlier, expected a `/RUN` command-line option to run as a service. The `TApplication` implementation of the registration routine does not allow this. There are several solutions to this problem:

1. Register the service manually, (or with an install program) and specify the `/RUN` (or some other) command-line option explicitly. The `StartService` function then just has to detect the presence of this command-line option.

2. Require a command-line option to start the program normally. This command-line can then be added to the shortcut for the program in the start menu. The disadvantage of this approach is that if the user uses the Windows Explorer and double-clicks on the program binary itself to start it, the application will be started (falsely) as a service.

3. Detect the username with which the application is being started. If this matches the username specified in the service registration (default the `LocalSystem` user), then the application is being started as a service. If the current username is not equal to the service user name, then the application is started manually. This is the approach used for instance in Borland's own socket server service. The disadvantage of this approach is that an application which must be run as an existing user (e.g. Administrator) will always decide to run as a service when it is started by this user. This should not be underestimated, because the Administrator is free to change the startup username after the service was installed.

Obviously, the first option is unambiguous and simple, and hence preferrable, but this requires 'external' registration of the service. The `StartService` function would then look as follows:

```
function StartService: Boolean;
begin
  Result := FindCmdLineSwitch('RUN',['-','/'], True);
end;
```

To show how it can be done using Borland's default implementation, the last option is used in the diskclean application. It can be done quite simply using the `TServiceManager` class presented earlier:

```
function StartService: Boolean;

Var
  Info : TServiceDescriptor;
  Buf : Array[0..255] of char;
  N : String;
  Size : DWord;
  F : TExt;

begin
  Result := False;
  Try
    With TServiceManager.Create(Nil) do
      try
        Connected:=True;
        QueryServiceConfig(SCleanService,Info);
        If CompareText(Info.UserName,'LocalSystem')=0 then
          Info.UserName:='System';
      finally
        Connected:=False;
        Free;
      end;
    Size:=256;
    GetUserName(@Buf[0],Size);
    N:=Buf;
    Result:=CompareText(N, Info.UserName)=0;
  Except
  end;
end;
```

As can be seen, the servicemanager class is used to fetch the username with which the application should run as a service. If the username is 'LocalSystem' (the default) then the name is changed to 'System', as that will be the name that is reported by the `GetUserName` call (a nice anomaly in itself). After this, the actual username is fetched using the `GetUserName` Windows API call, and the two values are compared. If they match, the application will start as a service.

10 Conclusion

Writing a complete service application is not hard to do. Using Delphi it is even more easily accomplished. Service applications can be used in a variety of tasks: the application presented here has its use - downloaded files and documents tend to clutter harddisks after some time - but can easily be changed to perform its task a little different - or to perform other tasks as well.