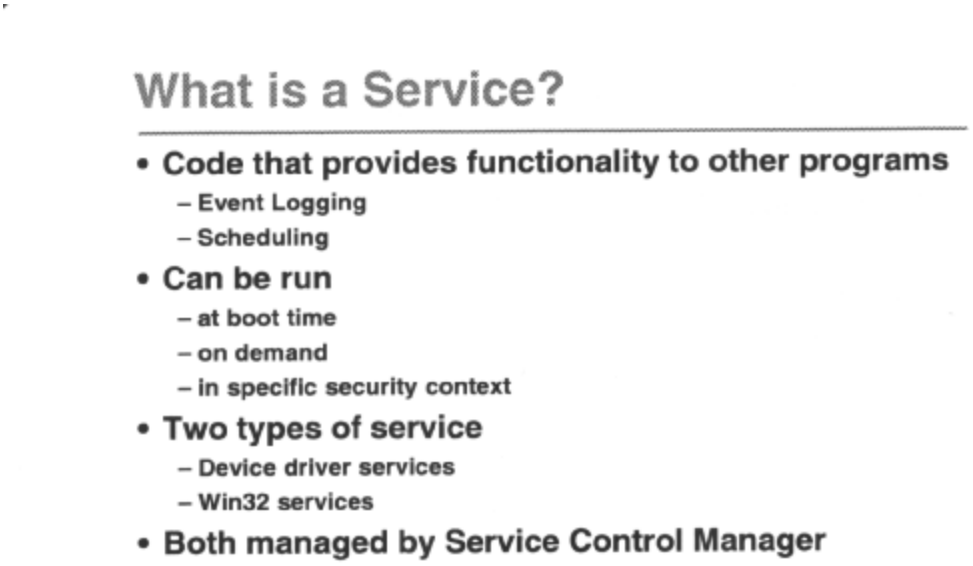


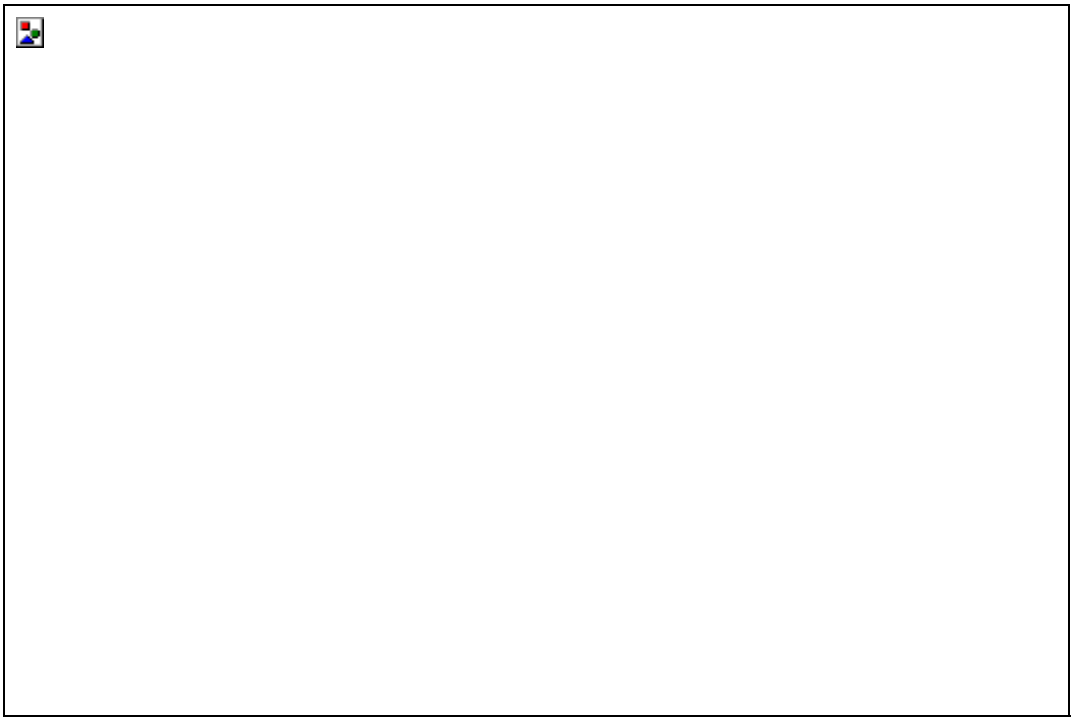
Services are programs. They are programs, which provide functionality to other executable code. Many processes in the Windows NT system are running as services. For example the *logon process* is run as a service. Services are different from ordinary programs in that they can be run (started) when the operating system is first booted.

Or they may be run on demand when another process requests their services. For example the *Remote Access Service* is not started until the user starts up the Remote Access dialer program. A service can be run in a specific security context, that of a user on the workstation or domain. Alternatively a service can be run under what is known as the Local System account. This is a low privilege account and most services are run in this security context.

There are two types of service on Windows NT. The first, *device driver services* such as disk drivers, file system drivers and display drivers, are beyond the scope of this course. The second, Win32 services are the subject of the rest of this chapter.

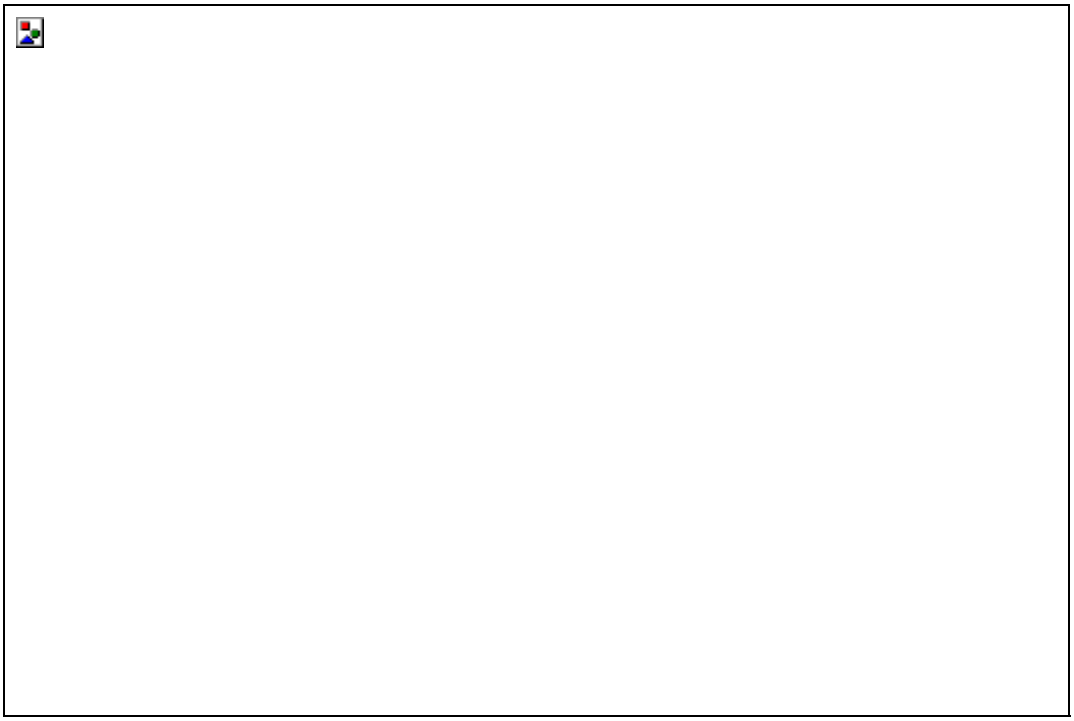
You can see which Win32 services are installed on your system, and whether they are started, stopped or paused by using the Services applet in the Control Panel. You can see which device driver services are installed by using the Devices-applet in the Control Panel

A process called the Service Control Manager manages both types of service. Control Panel is providing a user-interface for some of Service Control Manager's functionality.



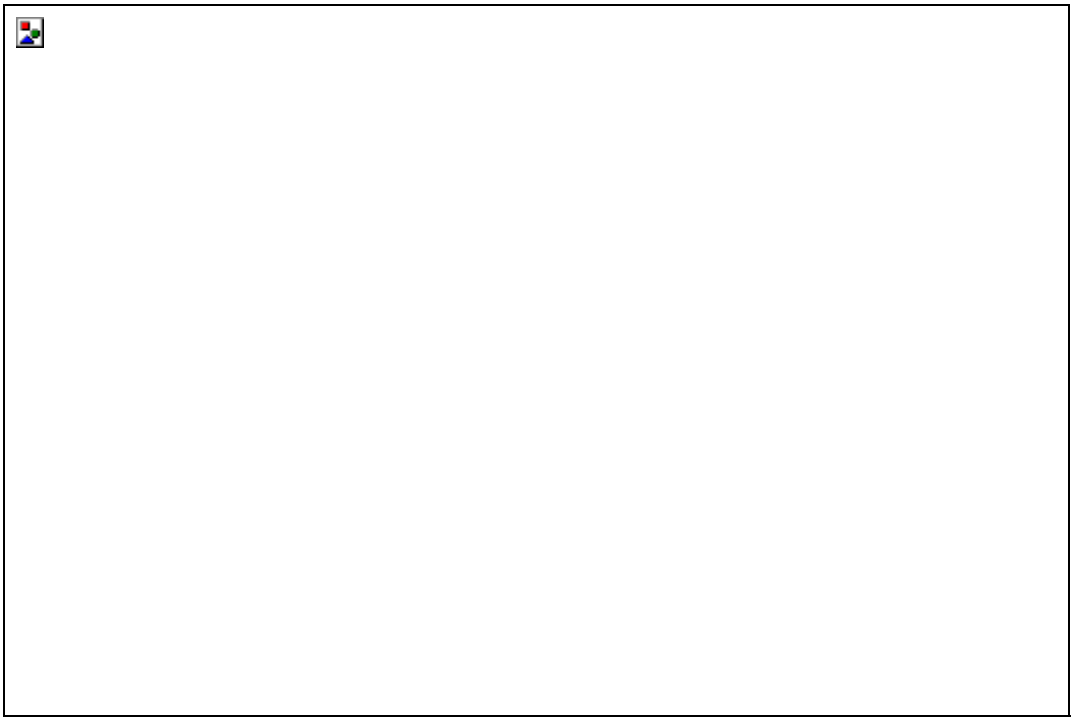
The *Service Control Manager* is responsible for managing all services on a Windows NT system, whether those services are device driver services or Win32 services. It starts services at system start time if required and on demand. It also maintains status information on services that are currently running. You can see this information by using the Services or Devices applets in Control Panel.

Lastly, *Service Control Manager* sends requests to a service, asking it to pause, continue or stop in response to user request or the request of another process in the system.



A straightforward service executable has a very simple `main()` function which calls `StartServiceCtrlDispatcher()`, passing the address of a `SERVICE_TABLE_ENTRY` as the parameter.
This structure contains the address of a `ServiceMain()` function.
Calling `StartServiceCtrlDispatcher()` calls the Service Control Manager which calls `ServiceMain()`.`ServiceMain()` must then install a `CtrlHandler()` function which is responsible for handling future requests from the *Service Control Manger*.

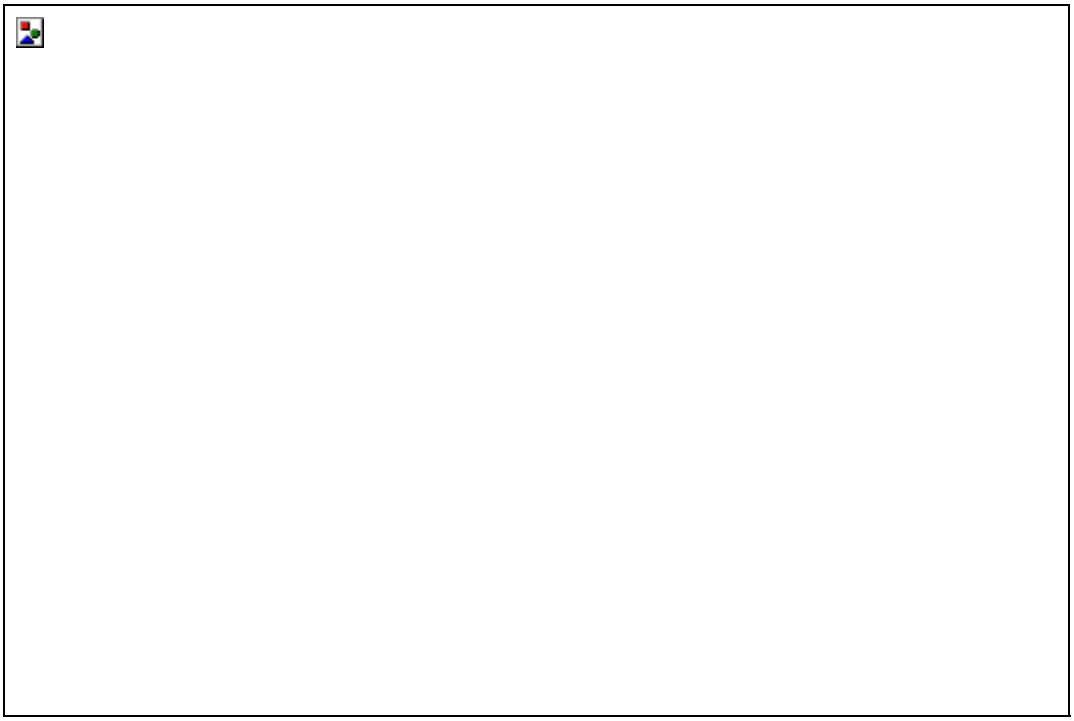
`ServiceMain()` then initializes the service and starts of the thread which will provide the service functionality. It is important to note that `ServiceMain()` must then wait for the service thread to end as when `ServiceMain()` exits the service is considered to be stopped.



The `main()` function for a simple service is very straightforward.
We fill in a single entry in a SERVICE_TABLE_ENTRY array, ensure the array is terminated with a NULL entry and call `StartServiceCtrlDispatcher()`.
The SERVICE_TABLE_ENTRY structure contains the name of the service and the address of the `ServiceMain()` function, RemoteDebugMain in this case.

An executable that is only supporting a single service, like the example above would be installed as a SERVICE_W1N32_OWN_PROCESS.

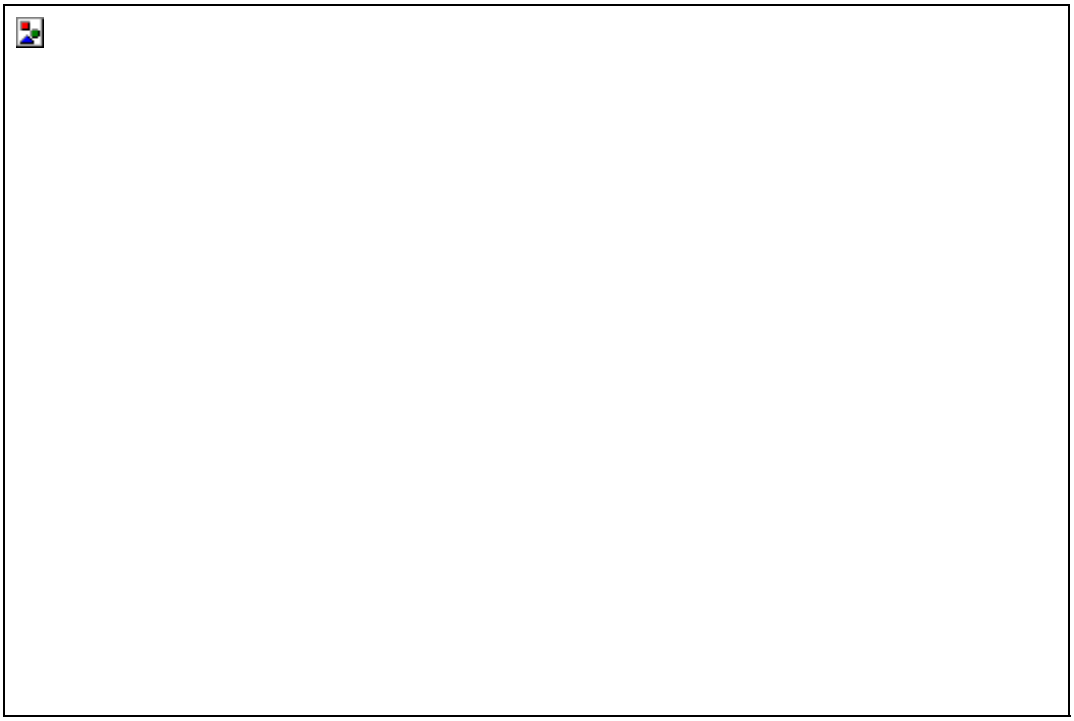
This tells Service Control Manager that the executable supports only a single service.
For a more complicated executable, one that supported several services from a single process, more entries would be made in the SERVICE_TABLE_ENTRY. For this type of service the installation program must specify that the service is SERVICE_W1N32_SHARE_PROCESS.

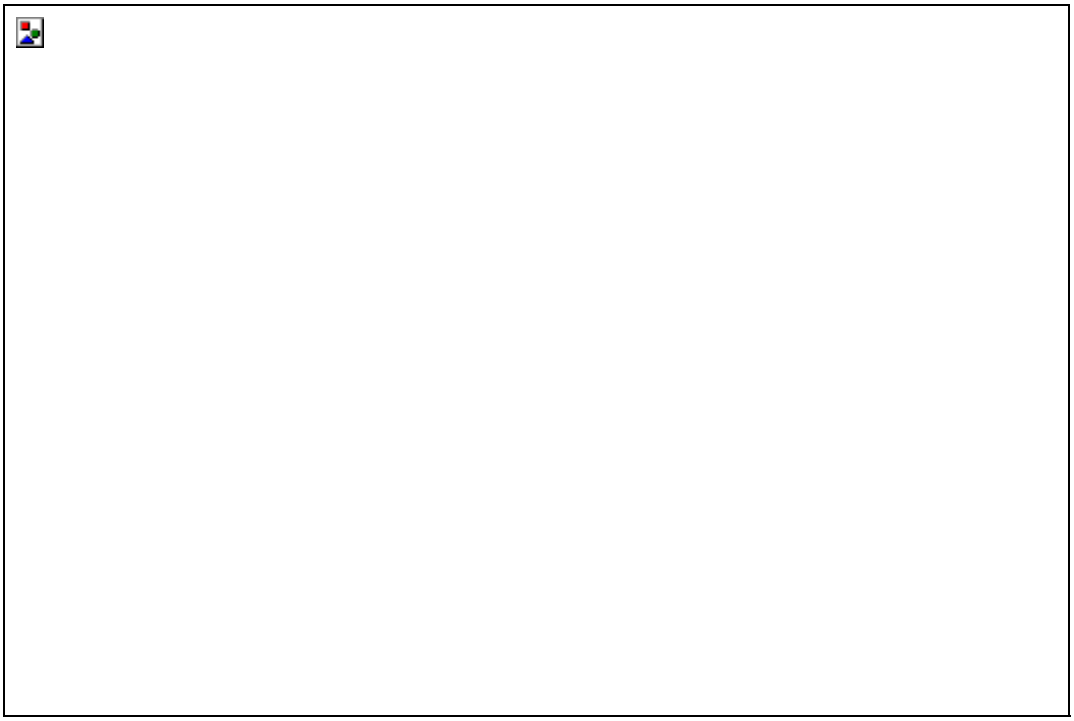


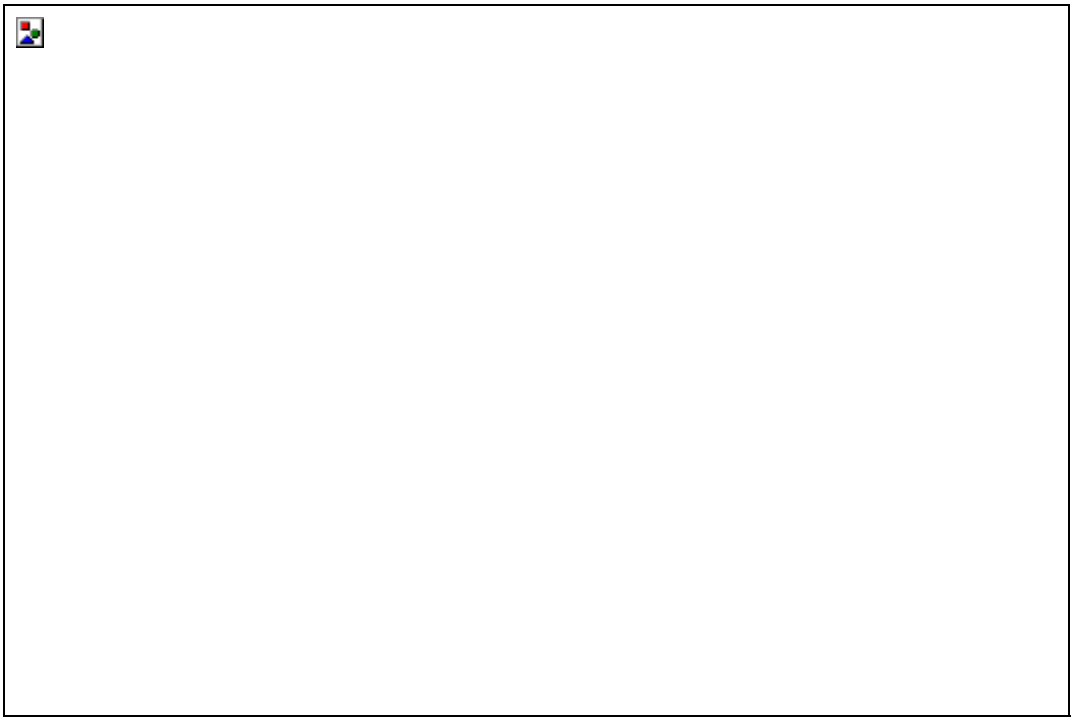
The `ServiceMain()` function is responsible for performing any initialization the service requires.
The first part of this initialization process is to register a `CtrlHandler()` with the *Service Control Manager* by calling `RegisterServiceCtrlHandler()`.
After registering the control handler `ServiceMain()` does any further initialization the service requires. While it does this it must periodically call `SetServiceStatus()` to let the *Service Control Manager* know how initialization is proceeding. `SetServiceStatus()` takes a SERVICE_STATUS_HANDLE which is returned by `RegisterServiceCtrlHandler()` and the address of a SERVICE_STATUS structure.

The SERVICE_STATUS structure tells the *Service Control Manager* several things; what the type of the service is, what the current state of the service is, which control requests will be accepted, exit codes for failure reporting, a check point and a wait hint.
The dwCheckPoint field should be incremented during service initialization after each operation.
This allows the user interface program (usually Control Panel) to give feedback to the user about how the service initialization is progressing.
This field is similarly used during a pending pause, continue or stop operation. When there is no pending operation this field should be zero.
The *dwWaitHint* field is used to give some idea of how long a pending operation will take to complete in milliseconds. If you have trouble starting your service, try increasing this value.
Again, while there is no pending operation, this field should be zero.

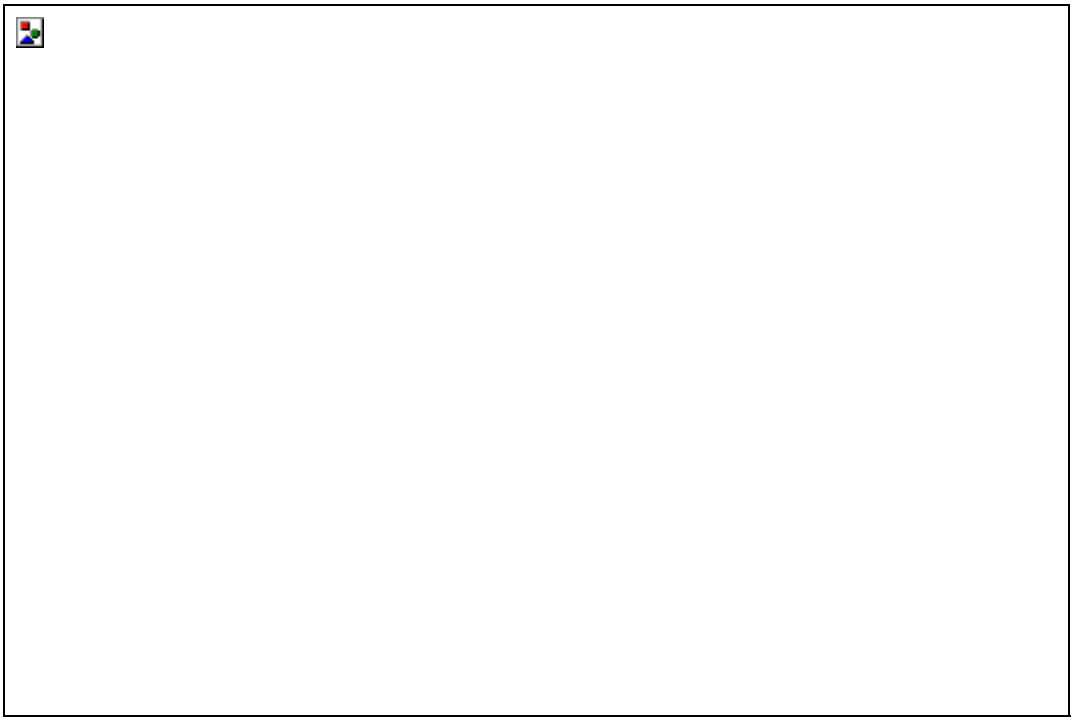
`ServiceMain()` then either provides the service itself or starts another thread to perform the service function. If `ServiceMain()` starts another thread it must block until that thread ends as when `ServiceMain()` exits the *Service Control Manager* considers the service to be stopped.







The `CtrlHandler()` function is responsible for responding to requests from the Service Control Manager. These requests may be SERVICE_CONTROL_PAUSE, SERVICE_CONTROL_CONTINUE, SERVICE_CONTROL_STOP or SERVICE_CONTROL_INTERROGATE. The `CtrlHandler()` must perform the requested action. After processing the request the `CtrlHandler()` must report the new status of the service by calling `SetServiceStatus()`
A service can specify in a call to `SetServiceStatus()` which control requests it will respond to. For example a service may choose not to respond to SERVICE_CONTROL_PAUSE (and hence SERVICE_CONTROL_CONTINUE **).**
A service **MUST** respond to SERVICE_CONTROL_INTERROGATE and this cannot be disabled.

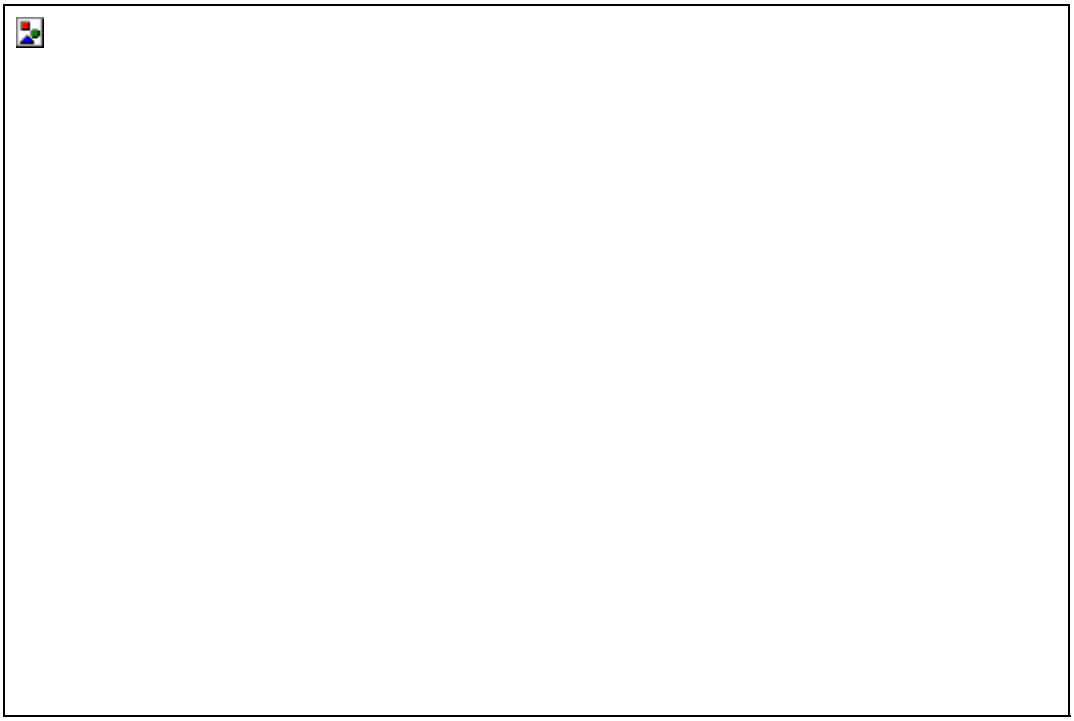


Unfortunately Windows NT does not ship with any utility to allow you to install your own services.
You have to write the install program too!
This install program must call `OpenSCManager()` which returns a handle to the *Service Control Manager*.
To install a new service GENERIC_WRITE access must be requested in the call the `OpenSCManager()`.

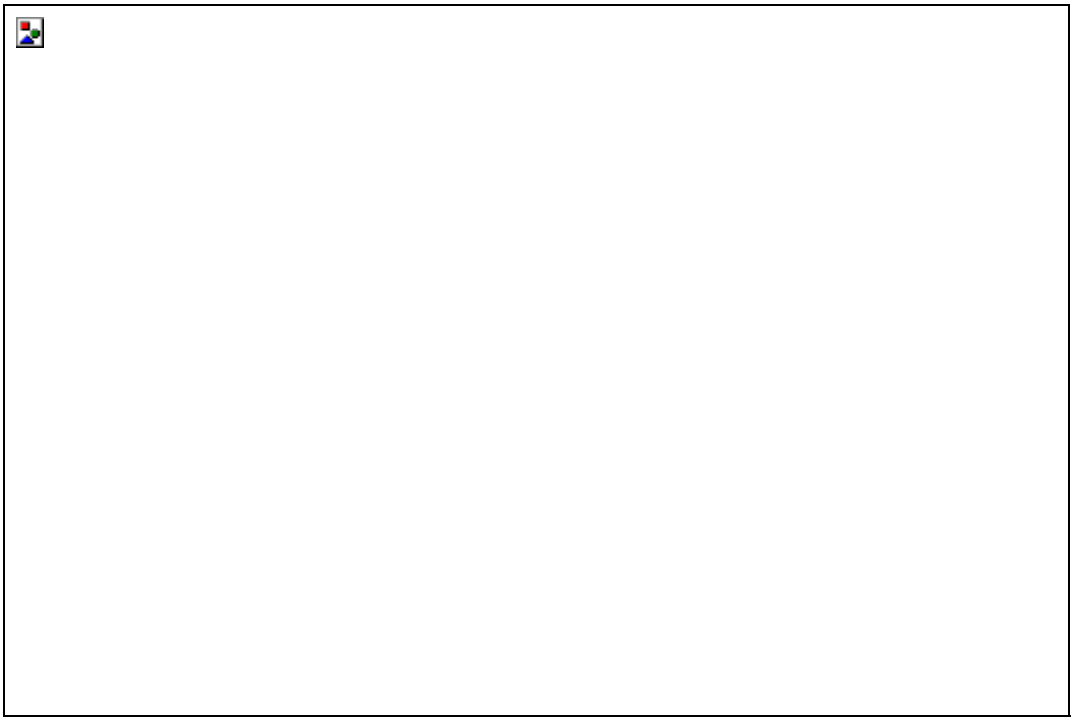
The returned handle is passed to a call to `CreateService()` which takes a large number of parameters including the path of the service executable, the name of the service that is displayed by user interface programs (such as Control Panel), whether the service starts at boot time or on demand, whether the service is a Win32 service or a device driver, which user account the service is going to run under...and many more. If `CreateService()` fails it returns NULL and `GetLastError()` will give the reason for the failure.

Otherwise `CreateService()` returns a handle to the service which can be used in calls to `APIs like QueryServiceConfig(), ChangeServiceConfig (), QueryServiceObjectSecurity() etc..`

Both the service handle returned by `CreateService()` and the *Service Control Manager* handle returned by `OpenSCManager()` should not be closed using the `CloseHandle()` API, rather `CloseServiceHandle()` should be used instead.



Usually a *service* will not have a user interface and by default services run on a separate virtual desktop that is invisible to the user. This means that while a service may create windows, dialogs etc, the user will never see them. Also any processes created by a service run on the same virtual desktop with the same results. Services that run in the security context of the system can intereact with the user desktop if the 'Allow service to interact with desktop' check box is selected in the *Services applet* in Control Panel. Note that this is not an option for services running in a user security context.



Event logging provides a consistent way of reporting error conditions, warnings and information to a system user or administrator. The operating system and applications can log events in the same way.

The operating system writes its event information into the *System event log*. Events that may appear in this log include low disk space warnings, information about network events, error reports on services that fail to start etc.

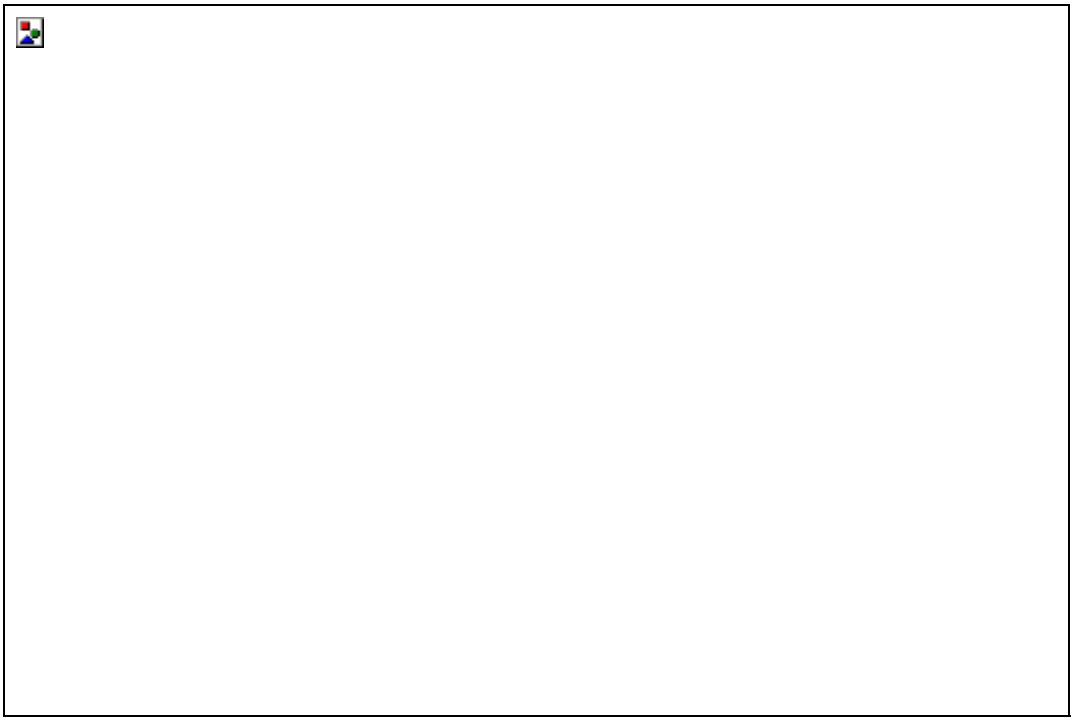
Security events are written into the security event log, these take the form of success or failure reports on certain events that are subject to security. Which events are logged can be specified using the User Manager application under the Policies, Audit... menu.

Applications write their event information into the application event log. An application can register its own message file (a resource DLL) by specifying a registry key under

```
HKEY_LOCAL_MACHINE

\SYSTEM\CurrentControlSet\Services\EventLog\Application
```

that specifies the message file path under the value EventMessageFile and the type of events supported (errors, warnings, information) under the TypesSupported value.

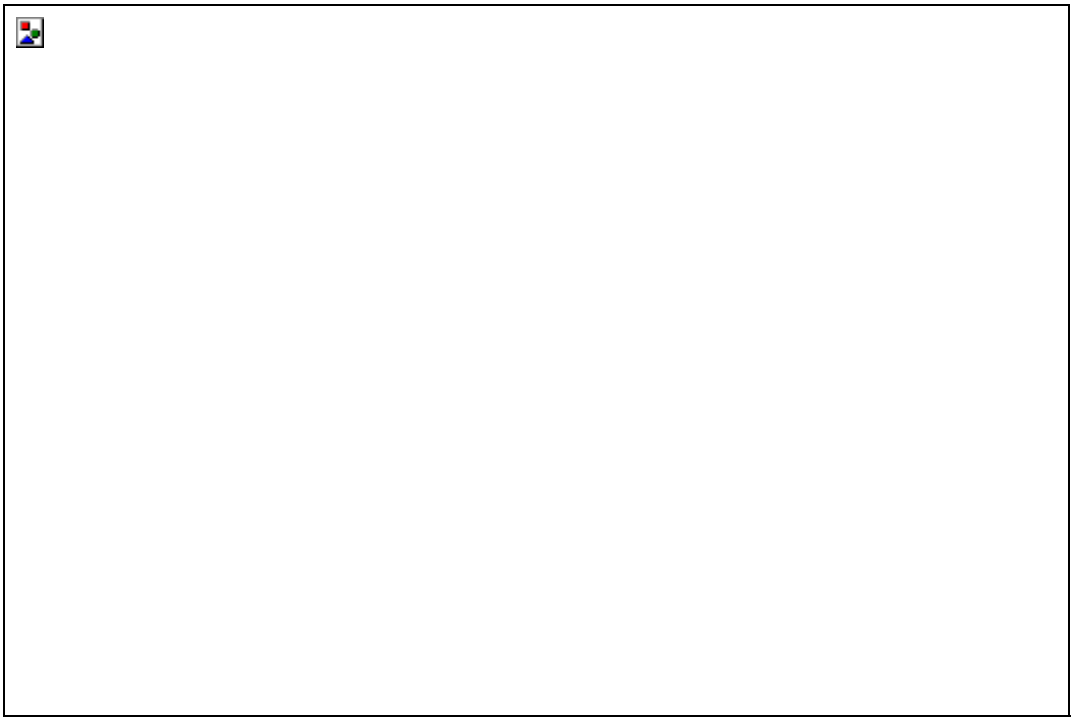


There are two ways of getting a handle to an event log. One is to call `OpenEventLog()` which returns a handle that can be used in calls the `ReadEventLog()` , `ClearEventLog()` `GetNumberOfEventLogRecord(),GetOldestEventLogRecord()` and `CloseEventLog()`.

This is the method used by the *Event Viewer application* when it displays the contents of an event log.
The second way is to call `RegisterEventSource()`.

This can be used to open an application specific event log (in terms of the message file used) or the default application event log.

The returned handle can be used in calls to `ReportEvent()` which writes event information to the event log.
When an application has finished reporting events it should close the handle using `DeregisterEventSource()`



Events are written to an event log using the `ReportEvent()` API. Parameters that must be specified include the type of event, the category (which is application specific) and the event ID that is used to retrieve a message string from the application message file.

It is also possible to specify the current users security ID, descriptive strings and binary data, though all these are optional.