Chapter 7

Adding New System Services to the Windows NT Kernel

IN THIS CHAPTER

+ Detailed Implementation of a system service
+ Adding new system services

CUSTOMIZING THE KERNEL for specific purposes has been very popular among developers long before Windows NT. Ancient Unix gurus and developers alike practiced the art. In Unix, for example, kernel developers can modify the kernel in several ways, such as adding new device drivers, kernel extensions, system calls, and kernel processes. In Windows NT, DDK provide means to add new device drivers. However, one of most effective ways of modifying the kernel — adding new system services to it — is *not* documented. This method proves more efficient than adding device drivers for several reasons discussed later in this chapter. Here, we focus on the detailed implementation of a system service inside the Windows NT kernel and explain, with examples, how *new* system services can add to the Windows NT.

In *Inside Windows NT*, Helen Custer mentions the design of system services and the possibility of adding new system services to the kernel:

> Using a system service dispatch table provides an opportunity to make native NT system services extensible. The kernel can support new system services simply by expanding the table without requiring changes to the system or to applications. After a code is written for a new system service, a system administrator could simply run a utility program that dynamically creates a new dispatch table. The new table will contain another entry that points to a new system service.

The capability to add new system services exists in Windows NT but it is *not* documented.

Very little changed between NT 3.51 and later versions of Windows NT in this area. The only thing being changed is that some of the data structures involved in implementation of a system service are located at the different offsets in the later versions of the operating system. We feel that our method of adding new system services may hold, possibly with very minor modifications, in future releases of Windows NT.

At the end of this chapter, we try to shed some light on the possible thought that went into the design of this portion of the operating system.

Detailed Implementation of a System Service in Windows NT

In Chapter 6, we discussed how a system service is invoked by the NTDLL.DLL at the request of the application. The SSDT (System Service Dispatch Table) and SSPT (System Service Parameter Table) help the kernel in accessing the right system service ID. The implementation of the SSDT and SSPT occurs similarly in all versions of Windows NT to date. We present the two implementations separately for clarity, one for Windows NT 3.51 and one for the later versions of the operating system such as Windows NT 4.0 and Windows 2000.

Below is the table containing the service ID mappings for all versions of Windows NT to date.

TABLE 7-1 **SERVICE ID MAPPINGS**

	KERNEL32 and ADVAPI32	USER32 and GDI32 Calls
Windows NT 3.51	Mapped to 0x0 through OxC3 service IDs inside NTOSKRNL *	Processed by the Win32 subsystem - a user mode process. No system services are provided in the kernel for handling these directly. These calls use the Win32 subsystem using kernel's LPC system services.
Windows NT 4.0 (up to Service Pack 5)	Mapped to 0x0 through OxD2 service IDs inside NTOSKRNL	Mapped to 0x1000 through 0x1 20A service IDs in the inside WIN32K.SYS. The kernel mode driver WIN32K.SYS takes over the functionality of the Win32 subsystem and supports these services.

	KERNEL32 and ADVAPI32	USER32 and GDI32 Calls
Windows NT 2000 (beta-2)	**0x0** through OxF3 service **IDs** inside NTOSKRNL	Mapped to 0x1000 through 0x1285 service IDs **in** the inside WIN32K.SYS. The kernel mode driver WIN32K.SYS takes over the functionality of the **Win32** subsystem **and** supports these services.

In Windows NT 3.51, only the KERNEL32 and ADVAPI32 functions of the operating system route through NTDLL.DLL to NTOSKRNL. The USER32 and ODD2 functions of the operating system implement as a part of the Win32 subsystem process (CSRSS). The USER32.DLL and GDI32.DLL provide wrappers, which calls the CSRSS process using the local procedure call (LPC) facility.

The functionality of USER32.DLL and GDI32.DLL is implemented differently in Windows NT 4.0 and Windows 2000. The functionality of the USER32 and GDI32 components is moved into the kernel mode driver WIN32K.SYS. The workhorse routines of NT 3.51's Win32 subsystem have transferred their load on the system services added by the addition of the WIN32K.SYS component. This explains why we see more system services versions later to Windows NT 3.51. This new set of system services corresponds to the USER32 and GDI32 components of the operating system.

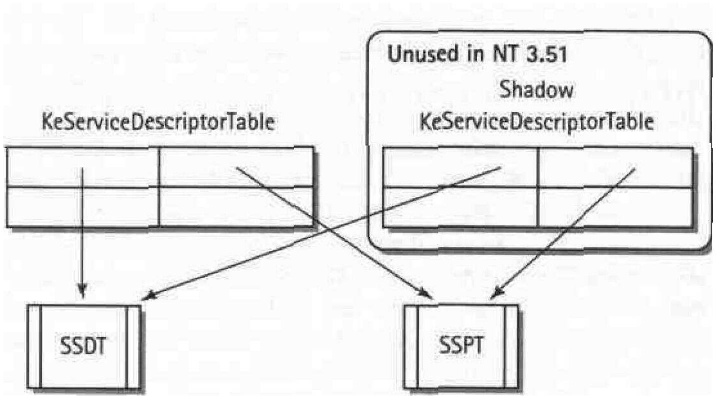

Figure 7-1: System **service** tables

Windows NT System Service Implementation

Here, we discuss the implementation of a system service under Windows NT. An INT 2Eh instruction implements the system services. The INT 2Eh handler is internally named as KiSystemService and hereafter we refer to it as the *handler*. Before entering the handler, the EAX register is loaded with the service ID and the EDX register with a pointer to the stack frame required for implementation of a particular service. The handler gets to the current TEB (Thread Environment Block) by looking at the Processor Control Region (PCR). The current TEB is stored at an offset of 0x124 in the Processor Control Region. The handler gets the address of the System Service Descriptor Table from the TEB. You can locate the address of the Service Descriptor Table at 0x124 offset in the TEB. Chapter 6 explains the format of the Service Descriptor Table.

The handler refers to the first entry in the Service Descriptor Table for service IDs less than 0x1000 and refers to the second entry of the table for service IDs greater than or equal to Ox 1000. The handler checks the validity of service IDs. If a service ID is valid, the handler extracts the addresses of the SSDT and SSPT. The handler copies the number of bytes (equal to the total number of bytes of the parameter list) described by the SSPT for the service-from user-mode stack to kernel-mode stack - and then calls the function pointed to by the SSDT for that service.

Initially, when any thread is started, the TEB contains a pointer to the Service Descriptor Table - identified internally as KeServiceDescriptorTable. KeService DescriptorTable contains four entries. Only the first entry in this table is used, which describes the service ids for some of the KERNEL32 and ADVAPI32 calls. Another Service Descriptor Table, internally named KeServiceDescriptor TableShadow, identically matches KeServiceDescriptorTable under NT 3.51. However, under later versions of the operating system, the second entry in the table is not NULL. The second entry points to another SSDT and SSPT. This SSDT and SSPT comprise part of the WIN32K.SYS driver. The WIN32K.SYS driver creates this entry during its initialization (in its DriverEntry routine) by calling the function called KeAddSystemServiceTable. (We provide more information on this later in this chapter.) This second entry describes the services exported by WIN32K.SYS for USER32 and GDI32 modules.

You should note that in all versions of Windows NT, KeServiceDescriptorTable contain only one entry and that all started threads point their TEBs to KeServiceDescriptorTable. This continues so long as the threads call services belonging to first entry in KeServiceDescriptorTable. When the threads call services above these limits (unlikely in 3.51, but very likely in later versions of Windows NT, because USER and GDI service IDs start with 0x1000), the KiSystemService jumps to a label JCiEndUnexpectedRange under NT 3.51 and JCiErrorMode under NT 4.0 and KiBBTEndUnexpectedRange in Windows 2000. Let's see what role the code at each label plays.

_KiEndUnexpectedRange (NT 3.51)

The following example shows the role of the code at the _KiEndUnexpectedRange label:

```
if (serviceID < 0x1000) {
/* It means if service id > 0xC3 and
 * service id < 0x1000
 */
return STATUS_INVALID_SYSTEM_SERVICE;
}
if (PsConvertToGuiThread() != STATUS_SUCCESS) {
return STATUS_INVALID_SYSTEM_SERVICE;
}

PsConvertToGuiThread()
{
if (PspW32ProcessCallout) {
/* In case of NT 3.51 this is code is never
 * invoked, since PspW32ProcessCallout is
 * always = 0
 */
/* This is only invoked for the later versions of the operating
system
 * Please refer to the next section for details
 */
} else {
return STATUS_ACCESS_DENIED;
}
}
```

_KiErrormode (in Windows NT 4.0 and KiBBTEndUnexpectedRange in Windows 2000)

The code resembles JCiEndUnexpectedRange, except that the PspW32Process Callout variable is always nonzero. Hence, the code in PsConvertToGuiThread proceeds further. It performs several tasks; we now describe the one of immediate interest.

PsConvertToGuiThread allocates a block of memory and copies KeService DescriptorTableShadow to the allocated block. Note that under NT 4.0 and Windows 2000, KeServiceDescriptorTableShadow contains two entries-one for KERNEL32 calls and one for USER32 and GDI32 calls. After copying this, the code updates the TEB of the current thread to point to this copy of KeService DescriptorTableShadow and then returns. This happens only the first time a

USER32 or GDI32 service is invoked. After this, all system services, including KERNEL32 module, route through this new table, since the first entry in this table already points to the SSDT and SSPT for the KERNEL32 functions.

KeServiceDescriptorTableShadow is not exported by the NTOSKRNL and therefore is a nonaccessible table.

Under Windows NT 3.51, both KeServiceDescriptorTable and the Shadow Table point to the same SSDT and SSPT and contain only one entry. Now, ask yourself this logical question: "Why do we have the Shadow Table at all when apparently it does not provide much help in NT 3.51?" We attempt to answer this question later in the chapter.

3 Note that once a process makes a USER32/GDI32 call, it permanently stops using the original KeServiceDescriptorTable and switches entirely to a copy of KeServiceDescriptorTableShadow.

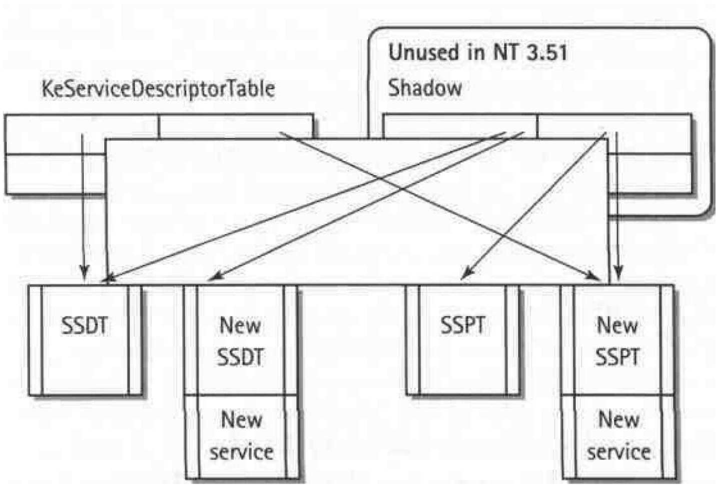

Figure 7-2: Adding new system services

Adding New System Services

Adding new system services involve the following steps:

1. Allocate a block of memory large enough to hold existing SSDT and SSPT and the extensions to each of the table.
2. Copy the existing SSDT and SSPT into this block of memory.

3. Append the new entries to the new copies of the two tables as shown in Figure 7-2.
4. Update KeServiceDescriptorTable and KeServiceDescriptorTableShadow to point to the newly allocated SSDT and SSPT.

In NT 3.51, because the Shadow Table is never used, you could get away without having to update it. In NT 4.0 and Windows 2000, however, the Shadow Table takes a leading role once a GDI32 or a USER32 call has been made. Therefore, it is important that you update both KeServiceDescriptorTable and KeService DescriptorTableShadow. If you fail to update KeServiceDescriptorTableShadow in NT 4.0 or Windows 2000, the newly added services will fail to work once a GDI32 or USER32 call is made. We recommend that you update both the tables in all versions of Windows NT so that you can use the same piece of code with all the versions of the operating systems.

One implementation issue in updating the KeServiceDescriptorTableShadow is that NTOSKRNL does not export this table. However, NTOSKRNL does export KeServiceDescriptorTable. So, how can you get the address of KeServiceDescriptor TableShadow?

The method we used for this is as follows. There is a function in NTOSKRNL called KeAddSystemServiceTable. This function is used by WIN32K.SYS driver for adding the USER32 and GDI 32 related functions. This function does refer to KeServiceDescriptorTableShadow. The first entry in both KeServiceDescriptorTable and KeServiceDescriptorTableShadow is the same. We iterate through each DWORD in the KeAddSystemServiceTable code, and for all valid addresses found in this function, we compare the 16 bytes (size of one entry in descriptor table) at this address with the first entry in KeServiceDescriptorTable. If we find the match, we consider that as the address of the KeServiceDescriptorTableShadow. This method seems to work in all Windows NT versions.

Example of Adding a New System Service

This example consists of three modules. One device driver contains the code for new system services and the mechanism of adding new system services to a Windows NT kernel. One DLL represents an interface to new system services (just as NTDLL.DLL provides interface for services called by KERNEL32.DLL). And one application links to this wrapper DLL and calls the newly added services. The newly added services print a debug message saying, "kernel service .... Called" and print the parameters passed to the services. Each service returns values 0, 1, and 2. The function AddServices() isolates the code for the mechanism of adding new system services.

Assuming first that the sample binaries are copied in C:\SAMPLES directory, here are the steps to try out the sample:

1. Run "instdrv extndsys c:\samples\extndsys.sys." This will install the extndsys.sys driver. The driver will add three new system services to Windows NT Kernel.
2. Run MYAPP.EXE. This will call wrapper functions in MYNTDLL.DLL to call newly added system services in EXTNDSYS.SYS.

```
#include "ntddk.h"
#include "stdarg.h"
#include "stdio.h"
#include "extnddrv.h"

#define DRIVER_SOURCE
#include "..\..\include\wintype.h"
#include "..\..\include\undocnt.h"


/* Prototypes for the services to be added */
NTSTATUS SampleService0(void);
NTSTATUS SampleService1(int param1);
NTSTATUS SampleService2(int param1, int param2);
/* TODO TODO TODO TODO
..............
..............
Add more to this list to add more services
*/


/* Table describing the new services */
unsigned int ServiceTableBase[]={(unsigned int)SampleService0,
                                 (unsigned int)SampleService1,
                                 (unsigned int)SampleService2,
/* TODO TODO TODO TODO
..............
..............
Add more to this list to add more services
*/
                                 };

/* Table describing the parameter bytes required for the new services */
unsigned char ParamTableBase[]={0,
                                4,
                                8,
/* TODO TODO TODO TODO
..............
..............
Add more parameter bytes to this list to add more services
*/
                               };



unsigned int *NewServiceTableBase; /* Pointer to new SSDT */
unsigned char *NewParamTableBase; /* Pointer to new SSPT */
unsigned int NewNumberOfServices; /* New number of services */

unsigned int StartingServiceId;

NTSTATUS SampleService0(void)
{
        trace(("Kernel service with 0 parameters called\n"));
        return STATUS_SUCCESS;
}

NTSTATUS SampleService1(int param1)
{
        trace(("Kernel service with 1 parameters called\n"));
        trace(("param1=%x\n", param1));
        return STATUS_SUCCESS+1;
}

NTSTATUS SampleService2(int param1, int param2)
{
        trace(("Kernel service with 2 parameters called\n"));
        trace(("param1=%x param2=%x\n", param1, param2));
        return STATUS_SUCCESS+2;
}

/* TODO TODO TODO TODO
..............
..............
Add implementations of other services here
*/

unsigned int GetAddrssofShadowTable()
```

```
{
        int i;
        unsigned char *p;
        unsigned int dwordatbyte;

        p=(unsigned char *)KeAddSystemServiceTable;

        for (i=0; i<4096; i++, p++) {
                __try {
                        dwordatbyte=*(unsigned int *)p;
                }
                __except (EXCEPTION_EXECUTE_HANDLER) {
                        return 0;
                }

                if (MmIsAddressValid((PVOID)dwordatbyte)) {
                        if (memcmp((PVOID)dwordatbyte,
&KeServiceDescriptorTable, 16)==0) {
                                if
((PVOID)dwordatbyte==&KeServiceDescriptorTable) {
                                        continue;
                                }
                                DbgPrint("Shadow @%x\n", dwordatbyte);
                                return dwordatbyte;
                        }
                }
        }
        return 0;
}


NTSTATUS AddServices()
{
    PServiceDescriptorTableEntry_t KeServiceDescriptorTableShadow;
    unsigned int NumberOfServices;

    NumberOfServices=sizeof(ServiceTableBase)/sizeof(ServiceTableBase[0]);

    trace(("KeServiceDescriptorTable=%x\n", &KeServiceDescriptorTable));


KeServiceDescriptorTableShadow=(PServiceDescriptorTableEntry_t)GetAddrssofShadow
Table();
```

```
    if (KeServiceDescriptorTableShadow==NULL) {
        return STATUS_UNSUCCESSFUL;
    }

    trace(("KeServiceDescriptorTableShadow=%x\n",
KeServiceDescriptorTableShadow));


NewNumberOfServices=KeServiceDescriptorTable.NumberOfServices+NumberOfServices;

    StartingServiceId=KeServiceDescriptorTable.NumberOfServices;

    /* Allocate sufficient memory to hold the existing services as well as
    the services you want to add */
    NewServiceTableBase=(unsigned int *) ExAllocatePool(PagedPool,
NewNumberOfServices*sizeof(unsigned int));
    if (NewServiceTableBase==NULL) {
        return STATUS_INSUFFICIENT_RESOURCES;
    }

    NewParamTableBase=(unsigned char *) ExAllocatePool(PagedPool,
NewNumberOfServices);
    if (NewParamTableBase==NULL) {
        ExFreePool(NewServiceTableBase);
        return STATUS_INSUFFICIENT_RESOURCES;
    }

    /* Backup the exising SSDT and SSPT */
    memcpy(NewServiceTableBase, KeServiceDescriptorTable.ServiceTableBase,
                KeServiceDescriptorTable.NumberOfServices*sizeof(unsigned
int));
    memcpy(NewParamTableBase, KeServiceDescriptorTable.ParamTableBase,
                KeServiceDescriptorTable.NumberOfServices);

    /* Append to it new SSDT and SSPT */
    memcpy(NewServiceTableBase+KeServiceDescriptorTable.NumberOfServices,
                ServiceTableBase, sizeof(ServiceTableBase));
    memcpy(NewParamTableBase+KeServiceDescriptorTable.NumberOfServices,
                ParamTableBase, sizeof(ParamTableBase));

    /* Modify the KeServiceDescriptorTableEntry to point to new SSDT and SSPT */
    KeServiceDescriptorTable.ServiceTableBase=NewServiceTableBase;
    KeServiceDescriptorTable.ParamTableBase=NewParamTableBase;
    KeServiceDescriptorTable.NumberOfServices=NewNumberOfServices;
    /* Also update the KeServiceDescriptorTableShadow to point to new SSDT and
SSPT */
    KeServiceDescriptorTableShadow->ServiceTableBase=NewServiceTableBase;
    KeServiceDescriptorTableShadow->ParamTableBase=NewParamTableBase;
    KeServiceDescriptorTableShadow->NumberOfServices=NewNumberOfServices;


    /* Return Success */
    DbgPrint("Returning success\n");
    return STATUS_SUCCESS;
}


NTSTATUS
DriverDispatch(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
    );

VOID
DriverUnload(
    IN PDRIVER_OBJECT DriverObject
    );


NTSTATUS
DriverEntry(
    IN PDRIVER_OBJECT  DriverObject,
    IN PUNICODE_STRING RegistryPath
    )
{
    MYDRIVERENTRY(L"extnddrv", FILE_DEVICE_EXTNDDRV, AddServices());
    return ntStatus;
}

NTSTATUS
DriverDispatch(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP           Irp
    )
{

    PIO_STACK_LOCATION  irpStack;
    PVOID               ioBuffer;
    ULONG               inputBufferLength;
    ULONG               outputBufferLength;
    NTSTATUS            ntStatus;

    Irp->IoStatus.Status      = STATUS_SUCCESS;
    Irp->IoStatus.Information = 0;

    irpStack = IoGetCurrentIrpStackLocation (Irp);

    switch (irpStack->MajorFunction)
    {
    case IRP_MJ_DEVICE_CONTROL:
        trace(("EXTNDDRV.SYS: IRP_MJ_CLOSE\n"));
        switch (irpStack->Parameters.DeviceIoControl.IoControlCode)
        {
                case IOCTL_EXTNDDRV_GET_STARTING_SERVICEID:

                trace(("EXTNDDRV.SYS:
IOCTL_EXTNDDRV_GET_STARTING_SERVICEID\n"));
                outputBufferLength = irpStack-
>Parameters.DeviceIoControl.OutputBufferLength;
                if (outputBufferLength<sizeof(StartingServiceId)) {
                        Irp->IoStatus.Status      =
STATUS_INSUFFICIENT_RESOURCES;
                } else {
                        ioBuffer     = (PULONG)Irp->AssociatedIrp.SystemBuffer;
                        memcpy(ioBuffer, &StartingServiceId,
sizeof(StartingServiceId));
                        Irp->IoStatus.Information = sizeof(StartingServiceId);
                }
                break;
        }
        break;
    }
    ntStatus = Irp->IoStatus.Status;

    IoCompleteRequest (Irp,
                       IO_NO_INCREMENT
                       );

    return ntStatus;
}
```

```
VOID
DriverUnload(
    IN PDRIVER_OBJECT DriverObject
    )
{
    WCHAR                  deviceLinkBuffer[]  = L"\\DosDevices\\EXTNDDRV";
    UNICODE_STRING         deviceLinkUnicodeString;


    RtlInitUnicodeString (&deviceLinkUnicodeString.
                          deviceLinkBuffer
                          );

    IoDeleteSymbolicLink (&deviceLinkUnicodeString);


    IoDeleteDevice (DriverObject->DeviceObject);

    trace(("EXTNDDRV.SYS: unloading\n"));
}

/* MYNTDLL.C
 * This DLL is a wrapper around the new services
 * added by the device driver. This DLL is like
 * NTDLL.DLL which is a wrapper around KERNEL32.DLL
 */
#include <windows.h>
#include <stdio.h>

#include <winioctl.h>
#include "..\sys\extnddrv.h"

typedef int NTSTATUS;

int ServiceStart;

__declspec(dllexport) NTSTATUS SampleService0(void)
{
_asm {
mov eax, ServiceStart
int 2eh
}
}
```

```
__declspec(dllexport) NTSTATUS
SampleService1(int param)
{
void **stackframe=&param;

_asm {
mov eax, ServiceStart
add eax, 1
mov edx, stackframe
int 2eh
}
}

__declspec(dllexport) NTSTATUS
SampleService2(int param1, int param2)
{
char **stackframe=&param1;

_asm {
mov eax, ServiceStart
add eax, 2
mov edx, stackframe
int 2eh
}
}

__declspec(dllexport) NTSTATUS
SampleService3(int param1, int param2, int param3)
{
char **stackframe=&param1;

_asm {
mov eax, ServiceStart
add eax, 3
mov edx, stackframe
int 2eh
}
}

__declspec(dllexport) NTSTATUS
SampleService4(int param1, int param2,
int param3, int param4)
{
char **stackframe=&param1;
```

```
_asm {
mov eax, ServiceStart
add eax, 4
mov edx, stackframe
int 2eh
}
}

__declspec(dllexport) NTSTATUS
SampleService5(int param1, int param2,
int param3, int param4,
int param5)
{
char **stackframe=&param1;

_asm {
mov eax, ServiceStart
add eax, 5
mov edx, stackframe
int 2eh
}
}

__declspec(dllexport) NTSTATUS
SampleService6(int param1, int param2,
int param3, int param4,
int param5, int param6)
{
char **stackframe=&param1;

_asm {
mov eax, ServiceStart
add eax, 6
mov edx, stackframe
int 2eh
}
}

BOOL SetStartingServiceId()
{
HANDLE   hDevice;
BOOL     ret;
```

```
hDevice = CreateFile (
"\\\\.\\extnddrv",
GENERIC_READ | GENERIC_WRITE,
0,
NULL,
OPEN_EXISTING,
FILE_ATTRIBUTE_NORMAL,
NULL
);

if (hDevice == ((HANDLE)-1))
{
                 MessageBox(0, "Unable to open handle to driver",
"Error", MB_OK);
ret = FALSE;
}
else
{
DWORD BytesReturned;
ret=DeviceIoControl(
hDevice,
IOCTL_EXTNDDRV_GET_STARTING_SERVICEID,
NULL,
NULL,
&ServiceStart,
sizeof(ServiceStart),
&BytesReturned,
NULL);

if (ret) {
if (BytesReturned!=sizeof(ServiceStart)) {
MessageBox(0, "DeviceIoControl failed",
"Error", MB_OK);
ret=FALSE;
} else {
ret = TRUE;
}
} else {
MessageBox(0, "DeviceIoControl failed",
"Error", MB_OK);
}
CloseHandle (hDevice);
}
```

```
return ret;
}

BOOL WINAPI
DllMain(HANDLE hModule,
DWORD Reason,
LPVOID lpReserved)
{
switch (Reason) {
case DLL_PROCESS_ATTACH:
        //
        // We're being loaded - save our handle
        //
return SetStartingServiceId();
default:
return TRUE;
}

}

/* This is a sample console application that calls
 * the newly added services. The services are called
 * through a wrapper DLL. The application simply
 * prints the return values from the newly added
 * system services.
 */

#include <windows.h>
#include <stdio.h>


#include "..\dll\myntdll.h"

main()
{
printf("SampleService0 returned = %x\n",
SampleService0());
printf("SampleService1 returned = %x\n",
SampleService1(0x10));
printf("SampleService2 returned = %x\n",
SampleService2(0x10, 0x20));

return 0;
}
```

Device Drivers as a Means of Extending the Kernel versus Adding New System Services

Writing pseudo device drivers and providing the DeviceloControl methods to the applications can also extend the kernel. However, in this case, each application that wants to use the DeviceloControl has to open a handle to the device, issue the DeviceloControl, and close the device. Extending the kernel by means of system services has its distinct advantages; first and foremost is that applications need not be aware of the device driver. Applications will just link to a DLL that provides an interface for the system services (just like NTDLL.DLL provides an interface for KERNEL32.DLL). Further, DeviceloContol proves much slower, especially if the DeviceloControl requires a large amount of data transfers between the application and the device driver. By using this technique of adding system services, you might write a set of system services and provide a user-level interface DLL that everybody can use. This implementation looks cleaner and more standardized than the DeviceloControl method.

KeAddSystemServiceTable

The WIN32K.SYS driver calls this function during its DriverEntry under Windows NT 4.0 and Windows 2000. This function looks somehow odd. The function expects five parameters: an index in the Service Descriptor Table where this new entry is to be added, SSDT, SSPT, the number of services, and one parameter for use only in checked build versions. This last parameter points to a DWORD Table that holds the value of the number of times each service gets called.

NT 3.51 Design versus NT 4.0 and Windows 2000 Design: Microsoft's Options

You might find it interesting to discover that the code manipulating the KeServiceDescriptorTableShadow resides in all versions of Windows NT - the only difference is that the code for allocating and copying the Shadow Table is not triggered under NT 3.51 based on the value of a PspW2ProcessCallout variable. This information might convince you that the relocation of USER32 and GDI32 component into the NT 4.0 and Windows 2000 kernel (as contrasted with the NT 3.51 kernel) is not only performance based-as Microsoft claims now-but something well thought out as an option when NT 3.51 was designed. This leads us to believe that Microsoft had two solutions implemented for USER32 and GDI32 modules-the LPC-based solution of using the Win32 subsystem and the INT 2Eh-based system service solution. Microsoft attempted the first solution under NT 3.51 and now settles for the second solution in later versions of Windows NT. The partial code for both solutions exists in NT 3.51, but there is no trace of the LPC solution for the Win32 subsystem under versions later than NT 3.51. So, we can also conclude that the future releases of NT, unless drastically different, will continue to use the INT 2Eh-based solution for WIN32K.SYS system services.

Summary

In this chapter, we discussed in detail the system service implementation of Windows NT. We explored some code fragments from a system service interrupt handler, using KiSystemServicefJ as an example. Next, we detailed the mechanism for adding new system services to the Windows NT kernel. We also used an example that adds three new system services to the Windows NT kernel. We compared extending the kernel with device drivers with extending the kernel by adding system services.

```
_asm {
mov eax, ServiceStart
add eax, 4
mov edx, stackframe
int 2eh
}
}

__declspec(dllexport) NTSTATUS
SampleService5(int param1, int param2,
int param3, int param4,
int param5)
{
char **stackframe=&param1;

_asm {
mov eax, ServiceStart
add eax, 5
mov edx, stackframe
int 2eh
}
}

__declspec(dllexport) NTSTATUS
SampleService6(int param1, int param2,
int param3, int param4,
int param5, int param6)
{
char **stackframe=&param1;

_asm {
mov eax, ServiceStart
add eax, 6
mov edx, stackframe
int 2eh
}
}

BOOL SetStartingServiceId()
{
HANDLE   hDevice;
BOOL     ret;
```

```
hDevice = CreateFile (
"\\\\.\\extnddrv",
GENERIC_READ | GENERIC_WRITE,
0,
NULL,
OPEN_EXISTING,
FILE_ATTRIBUTE_NORMAL,
NULL
);

if (hDevice == ((HANDLE)-1))
{
                  MessageBox(0, "Unable to open handle to driver",
"Error", MB_OK);
ret = FALSE;
}
else
{
DWORD BytesReturned;
ret=DeviceIoControl(
hDevice,
IOCTL_EXTNDDRV_GET_STARTING_SERVICEID,
NULL,
NULL,
&ServiceStart,
sizeof(ServiceStart),
&BytesReturned,
NULL);

if (ret) {
if (BytesReturned!=sizeof(ServiceStart)) {
MessageBox(0, "DeviceIoControl failed",
"Error", MB_OK);
ret=FALSE;
} else {
ret = TRUE;
}
} else {
MessageBox(0, "DeviceIoControl failed",
"Error", MB_OK);
}
CloseHandle (hDevice);
}
```

```
return ret;
}

BOOL WINAPI
DllMain(HANDLE hModule,
DWORD Reason,
LPVOID lpReserved)
{
switch (Reason) {
case DLL_PROCESS_ATTACH:
        //
        // We're being loaded - save our handle
        //
return SetStartingServiceId();
default:
return TRUE;
}

}

/* This is a sample console application that calls
 * the newly added services. The services are called
 * through a wrapper DLL. The application simply
 * prints the return values from the newly added
 * system services.
 */

#include <windows.h>
#include <stdio.h>


#include "..\dll\myntdll.h"

main()
{
printf("SampleService0 returned = %x\n",
SampleService0());
printf("SampleService1 returned = %x\n",
SampleService1(0x10));
printf("SampleService2 returned = %x\n",
SampleService2(0x10, 0x20));

return 0;
}
```

Device Drivers as a Means of Extending the Kernel versus Adding New System Services

Writing pseudo device drivers and providing the DeviceloControl methods to the applications can also extend the kernel. However, in this case, each application that wants to use the DeviceloControl has to open a handle to the device, issue the DeviceloControl, and close the device. Extending the kernel by means of system services has its distinct advantages; first and foremost is that applications need not be aware of the device driver. Applications will just link to a DLL that provides an interface for the system services (just like NTDLL.DLL provides an interface for KERNEL32.DLL). Further, DeviceloContol proves much slower, especially if the DeviceloControl requires a large amount of data transfers between the application and the device driver. By using this technique of adding system services, you might write a set of system services and provide a user-level interface DLL that everybody can use. This implementation looks cleaner and more standardized than the DeviceloControl method.

KeAddSystemServiceTable

The WIN32K.SYS driver calls this function during its DriverEntry under Windows NT 4.0 and Windows 2000. This function looks somehow odd. The function expects five parameters: an index in the Service Descriptor Table where this new entry is to be added, SSDT, SSPT, the number of services, and one parameter for use only in checked build versions. This last parameter points to a DWORD Table that holds the value of the number of times each service gets called.

NT 3.51 Design versus NT 4.0 and Windows 2000 Design: Microsoft's Options

You might find it interesting to discover that the code manipulating the KeServiceDescriptorTableShadow resides in all versions of Windows NT - the only difference is that the code for allocating and copying the Shadow Table is not triggered under NT 3.51 based on the value of a PspW2ProcessCallout variable. This information might convince you that the relocation of USER32 and GDI32 component into the NT 4.0 and Windows 2000 kernel (as contrasted with the NT 3.51 kernel) is not only performance based-as Microsoft claims now-but something well thought out as an option when NT 3.51 was designed. This leads us to believe that Microsoft had two solutions implemented for USER32 and GDI32 modules-the LPC-based solution of using the Win32 subsystem and the INT 2Eh-based system service solution. Microsoft attempted the first solution under NT 3.51 and now settles for the second solution in later versions of Windows NT. The partial code for both solutions exists in NT 3.51, but there is no trace of the LPC solution for the Win32 subsystem under versions later than NT 3.51. So, we can also conclude that the future releases of NT, unless drastically different, will continue to use the INT 2Eh-based solution for WIN32K.SYS system services.

Summary

In this chapter, we discussed in detail the system service implementation of Windows NT. We explored some code fragments from a system service interrupt handler, using KiSystemServicefJ as an example. Next, we detailed the mechanism for adding new system services to the Windows NT kernel. We also used an example that adds three new system services to the Windows NT kernel. We compared extending the kernel with device drivers with extending the kernel by adding system services.