Chapter 2

# Writing Windows NT Device Drivers

IN THIS CHAPTER

- Prerequisites to getting started
- The building procedure
- The structure of a device driver

Most OF THE SAMPLES IN this book are Windows NT kernel mode device drivers. This chapter contains the information you need to build device drivers and understand the samples in this book. This chapter is *not* a complete guide to writing device drivers. The best sources of information for detailed coverage of the topic are Art Baker's *The Windows NT Device Driver Book: A Guide for Programmers* and the documentation that ships with the Windows NT Device Driver Kit (DDK).

## Prerequisites to Writing NT Device Drivers

You must install the following tools to create a working development environment for Windows NT kernel mode device drivers:

Windows NT Device Driver Kit (DDK) from Microsoft For the development of device drivers, you need to install the Device Driver Kit on your machine. The Device Driver Kit is available with the MSDN Level 2 subscription. The kit consists of sets of header files, libraries, and tools that enable easy development of device drivers.

32-bit compiler You need a 32-bit compiler to compile the device drivers. We strongly recommend using the Microsoft compiler to build the samples in this book.

Win32 Software Development Kit (SDK) Although it is not necessary for compiling the samples from this book, we recommend installing the latest version of the Win32 SDK on your machine. Also, when you build device drivers using

the DDK tools, you should set the environment variable MSTOOLS to point to the location where the Win32 SDK is installed. You can fake the installation of the Win32 SDK by adding the environment variable MSTOOLS with the System applet in the Control Panel.

# Driver Build Procedure

The Windows NT 4.0 Device Driver Kit installation adds four shortcuts to the Start menu: Free Build Environment, Checked Build Environment, DDK Help, and Getting Started. The Free Build Environment and Checked Build Environment shortcuts both refer to a batch file called SETENV.BAT, but have different command line arguments. Assuming that the DDK is installed in directory E:\DDK40, the Free Build Environment shortcut refers to this command line:

```
%SystemRoot%\System32\cmd.exe /k E:\DDK40\bin\setenv.bat
E:\DDK40 free
```

The Checked Build Environment shortcut, on the other hand, refers to this command line:

```
%SystemRoot%\System32\cmd.exe /k E:\DDK40\bin\setenv.bat E:\DDK40
checked
```

Both shortcuts spawn CMD.EXE and ask it to execute the SETENV.BAT file with appropriate parameters. After executing the command, CMD.EXE still keeps running because of the presence of the /k switch. The SETENV.BAT file sets the environment variables, which are added to the CMD.EXE process's environment variable list. The DDK tools, which are spawned from CMD.EXE, refer to these environment variables. SETENV.BAT sets the environment variables, including BUILD_DEFAULT, BUILD_DEFAULT_TARGETS, BULLD_MAKE_PROGRAM, and DDKBUILDENV.

The drivers are compiled using the utility called BUILD.EXE, which is shipped with the DDK. This utility takes as input a file named SOURCES. This file contains the list of source files to be compiled to build the driver. This file also contains the name of the target executable, the type of the target executable (for example, DRIVER or PROGRAM), and the path of the directory where the target executable is to be created.

Each sample device driver included with the DDK contains a makefile. However, this is not the actual makefile for the device driver sample. Instead, the makefile for each sample device driver includes a common makefile, named MAKEFILE.DEF, which is present in the INC directory of the DDK installation directory.

Here is the sample makefile from the DDK sample:

```
#
# DO NOT EDIT THIS FILE!!!  Edit .\sources. if you want to add a
new source
# file to this component.  This file merely indirects to the real
make file
# that is shared by all the driver components of the Windows NT DDK
#
!INCLUDE $(NTMAKEENV)\makefile.def
```

Some of the driver samples in this book have Assembly language files (.ASM files). You cannot refer to the .ASM file directly into the SOURCES file. Instead, you have to create a directory called 1386 in the directory where the source files for the drivers are kept. All the .ASM files for the drivers must be kept in the 1386 directory. The BUILD.EXE utility automatically uses ML.EXE to compile these .ASM files.

BUILD.EXE generates the appropriate driver or application based on the settings specified in the SOURCES file and using the platform-dependent environment variables. If there are any errors during the BUILD process, the errors are logged to a file called as BUILD.ERR. If there are any warnings, they are logged to the BUILD.WRN file. Also, the BUILD utility generates a file called BUILD.LOG, which contains lists of commands invoked by the BUILD utility and the messages given by these tools.

# Structure of a Device Driver

Just as every Win32 application has an entry point (main/WinMain), every kernel mode device driver has an entry point called DriverEntry. A special process called SYSTEM loads the device drivers. Hence, the DriverEntry of each device driver is called in the context of the SYSTEM process. Each device driver is represented by a device name in the system, so each driver has to create a device name for its device. This is done with the loCreateDevice function. If Win32 applications need to open the handle to a device driver, the driver needs to create a symbolic link for its device in the DosDevices object directory. This is done using a call to loCreateSymbolicLink. Typically, in the DriverEntry routine of a device driver, the device object and the symbolic link object are created for a device and some driver or device-specific initialization is performed.

Most of the device driver samples in this book involve pseudo device drivers. These drivers do not control any physical device. Instead, they complete tasks that can be performed only from the device driver. (The device driver runs at the most privileged mode of the processor - Ring 0 in Intel processors.) In addition, the DriverEntry is supposed to provide sets of entry points for other functions, such as OPEN, CLOSE, DEVICEIOCONTROL, and so on. These entry points are provided by filling in some fields in the device object, which is passed as a parameter to the DriverEntry function.

Because most of the drivers in this book are pseudo device drivers, the DriverEntry routine is the same for all of them. Only the device driver-specific initialization is different. Instead of repeating the same piece of code in each of the driver samples, a macro is written. The macro is called **MYDRIVERENTRY**:

```
#define MYDRIVERENTRY(DriverName,DeviceId,DriverSpecificInit)
PDEVICE_OBJECT deviceObject=NULL;
NTSTATUS ntStatus;
WCHAR deviceNameBuffer[]=L"\\Device\\"##DriverName;
UNICODE_STRING deviceNameUnicodeString;\
WCHAR deviceLinkBuffer[]=L"\\DosDevices\\"##DriverName;
UNICODE_STRING deviceLinkUnicodeString;
RtlInitUnicodeString(&deviceNameUnicodeString,
deviceNameBuffer);
ntStatus = IoCreateDevice(DriverObject,
                0,
&deviceNameUnicodeString,
##DeviceId,
0,
TRUE,
&deviceObject);

if (NT_SUCCESS(ntStatus)){
RtlInitUnicodeString(&deviceLinkUnicodeString,
deviceLinkBuffer);
ntStatus= IoCreateSymbolicLink(
&deviceLinkUnicodeString,
&deviceNameUnicodeString);
if (!NT_SUCCESS(ntStatus)) {
IoDeleteDevice (deviceObject);
return ntStatus;
}

ntStatus=##DriverSpecificInit;
if (!NT_SUCCESS(ntStatus)) {
IoDeleteDevice (deviceObject);
IoDeleteSymbolicLink(&deviceLinkUnicodeString);
return ntstatus;
}
                \
DriverObject->MajorFunction[IRP_MJ_CREATE] =
   DriverObject->MajorFunction[IRP_MJ_CLOSE] =
   DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] =
DriverDispatch;
```

```
DriverObject->DriverUnload=DriverUnload;
return STATUS_SUCCESS;

} else {                                                    return
ntStatus;
```

The macro takes the following three parameters:

+ The first parameter is the name of the driver, which will be used for creating the device name and symbolic link.
+ The second parameter is the device ID, which uniquely identifies the device.
+ The third parameter is the name of the function, which contains the driver-specific initialization.

The macro expands into calling the necessary functions such as loCreateDevice and loCreateSymbolicLink. If these functions succeed, the driver calls the driver-specific initialization function specified by the third parameter. If the function returns failures, the macro returns the error code of the specific initialization function. If the function succeeds, the macro fills in various function pointers for other functions supported by the driver in the DriverObject. Once this macro is used in the DriverEntry function, you need to write the DriverDispatch and DriverUnload functions, as the macro refers to these functions.

The macro definition can be found in UNDOCNT.H on the included CD-ROM.

All the requests to device driver are sent in the form of an I/O Request packet (IRP). The driver expects the system to call the specific driver function for all device driver requests based on the function pointers filled in during DriverEntry. We assume that all the driver functions are filled in with the address of the DriverDispatch function in the following discussion.

The DriverDispatch function is called with an IRP containing the command code of IRP_MJ_CREATE whenever an application opens a handle to a device driver using the CreateFile API call. The DriverDispatch function is called with an IRP containing the command code of IRP_MJ_CLOSE whenever an application closes its handle to a device driver using the CloseHandle API function. The DriverDispatch function is called with an IRP containing the command code of IRP_MJ_DEVICE_CONTROL whenever the application uses the DeviceloControl API function to send or receive data from a device driver. If the driver functionality is being used by multiple processes, the driver can use the CREATE and CLOSE entry points to perform per-process initialization.

Because all these requests end up calling DriverDispatch, you need to have a way to identify the actual function requested. You can accomplish this by looking at the

MajorFunction field in an I/O Request Packet (IRP). The request packet contains the function code and any other additional parameters required to complete the request. The DriverUnload routine is called when the device driver is unloaded from the system. Just like DriverEntry, the DriverUnload function is called in the context of the SYSTEM process. Typically, in a DriverUnload routine, the device driver deletes the symbolic link and the device name created during DriverEntry and performs some device-specific uninitialization.

# Summary

In this chapter, we covered the software requirements for building Windows NT device drivers, the procedure for building device drivers, and the structure of a typical device driver. Along the way, we explained a simple macro that you can use to generate the driver entry code for a typical device drive.