

Creating a WinCE NDIS Device Driver

By Michael Migdol

It hasn't taken long for Microsoft's® Windows CE® (WinCE) OS to become a key player in the embedded OS market. The speed at which the OS has risen in popularity has left both network chip developers and systems integrators scrambling to produce software drivers for their products. Although Microsoft touts the NDIS interface in WinCE as a way to provide an OS-independent API for network device drivers, developing any driver for WinCE is certainly non-trivial. WinCE currently supports only NDIS Ethernet and IrDA miniport drivers, but several factors make writing even these simple NDIS drivers for WinCE somewhat complicated.

First of all, WinCE NDIS device drivers follow a different model from the traditional stream interface drivers that are most typical for Windows CE. NDIS device drivers are loaded differently and handle interrupt requests differently from other types of device drivers. More importantly, the WinCE version of NDIS lacks some of the functions needed for certain types of network drivers, particularly those utilizing Direct Memory Access (DMA). The process by which WinCE NDIS device drivers are built and installed present another complication.

This article discusses some of the issues a developer of WinCE NDIS device drivers can expect to encounter. We will start with a basic overview of the WinCE Device Manager, focusing on driver loading, WinCE interrupt handling, and WinCE memory management. In each case we will start by covering the topic in general, and then discuss specifics for NDIS device drivers. Next, we'll talk about how your driver will get built and installed onto the target device. Finally, we'll conclude with a simple discussion of exactly what steps are required to port an existing NDIS device driver from another OS.

This article alone won't give you quite enough information to write a WinCE driver if you've never been exposed to NDIS before. If this is the case, I would suggest that immediately **after** reading this article, you refer to the "Network Drivers" section of the Windows 2000 DDK documentation. This article should provide enough context so that on reading that documentation, you will understand how the information relates to WinCE.

Just to warn you in advance — we won't be discussing WinCE 3.0 specifics, NDIS drivers outside the scope of NDIS miniports, power management, or NT Embedded. Some of the alternate information sources presented at the end of the article discuss these topics if you are interested.

WinCE NDIS Device Driver Basics

This section discusses how WinCE NDIS device drivers are loaded and how they handle interrupts. In each case, we'll look at WinCE device drivers in general as well as the specific case of WinCE NDIS drivers.

How Device Drivers Get Loaded

Figure 1 shows the WinCE Device Driver hierarchy and which drivers are loaded by whom. At the top, we have the device drivers GWES.EXE (Graphical Windowing and Event Subsystem) and DEVICE.EXE (also known as the Device Manager). These two drivers are loaded very early in the boot process by the

kernel. Native drivers whose operation is integral to the function of the OS are loaded by GWES.EXE and DEVICE.EXE. Stream Interface Drivers for devices such as serial ports are typically loaded by DEVICE.EXE in one of two ways. Drivers for built-in devices are loaded if they are included in the HKEY_LOCAL_MACHINE\Drivers\Builtin registry key. On the other hand, PC Card device drivers can be loaded using registry entries based on either their device Plug-and-Play identifier or with a separate detection function; this will be discussed in more detail below.

NDIS device drivers are loaded by a parent device driver contained in the NDIS.DLL dynamically linked library, or DLL. I will refer to NDIS.DLL from now on as the NDIS "Wrapper" to distinguish it from the NDIS Miniport drivers that are the focus of this article. The NDIS Wrapper is part of the Run-Time Library provided by Microsoft, and is actually a stream interface driver. This driver can be loaded either as a built-in device or as a PC-Card add-on device as discussed above.

When loaded as a "built-in device", the NDIS Wrapper scans the HKEY_LOCAL_MACHINE\COMM registry key for NDIS drivers, loading each in turn. Unlike NDIS drivers under other operating systems, WinCE NDIS drivers are always loaded as a DLL. As with all DLLs, they must export an entry function (usually called *DLLEntry*) that is called by the OS when the DLL file is loaded or unloaded. *DLLEntry* for NDIS drivers typically only prints out debug messages that give some indication that the driver was, in fact, loaded.

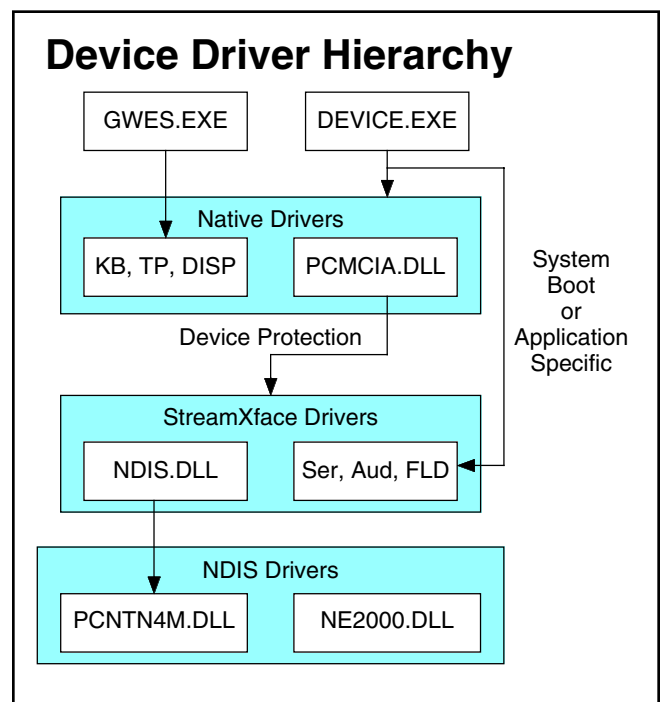


Figure 1

After the driver DLL has been loaded, the NDIS Wrapper calls the driver's initialization function called *DriverEntry*. The NDIS wrapper makes calls to the functions provided by the driver through a function table. *DriverEntry* normally sets up the NDIS data structure that contains this information, and then calls *NdisMRegisterMiniport* to register this information with the NDIS Wrapper. Figure 2 illustrates this relationship.

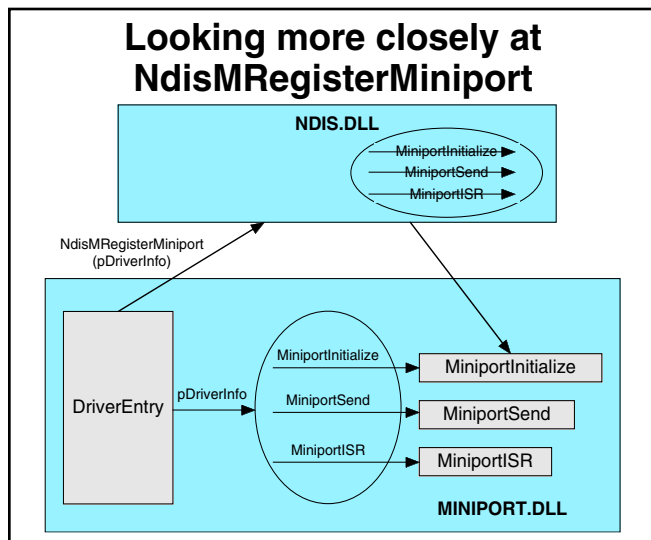


Figure 2

Drivers for PCMCIA cards are slightly more complex. A native driver, PCMCIA.DLL, provides the interface to the socket itself. The Device Manager (DEVICE.EXE) loads PCMCIA.DLL and then requests notification whenever a new PCMCIA device is inserted. When this occurs, the Device Manager then queries the PCMCIA device driver to find the card's device identifier (also known as the logical device ID) from the card's PCMCIA attribute space. This identifier is then compared against the subkeys of the registry key HKEY_LOCAL_MACHINE\Drivers\PCMCIA for a match, loading the driver in that sub-key if it finds one. For network devices, the driver loaded will be the NDIS wrapper, NDIS.DLL.

If no match is found, the detection functions listed in the HKEY_LOCAL_MACHINE\Drivers\PCMCIA\Detect registry key are called. If one of these detection functions returns a positive response, the driver indicated in that registry key will be loaded. Again, this will be NDIS.DLL for network devices.

The detection functions are handy if you have a network device that uses a common programming interface such as NE2000 or LANCE. In these cases, devices with many different device identifiers will be able to use the same device driver. We'll talk more about how registry keys can be added to the OS a little later. Details on the contents of the registry are beyond the scope of this paper, but can be found in the WinCE DDK documentation.

How Interrupts are Handled

Interrupts under WinCE follow a complex path. Typically, hardware devices assert an interrupt signal input of a programmable interrupt controller. The interrupt controller generates a vector that is used as an index into the operating system's "Interrupt Vector Table."

In WinCE, the OS function *HookInterrupt* is used to assign an Interrupt Service Routine (ISR) to any given interrupt vector. ISRs are run at kernel level in WinCE; and within the ISR, interrupts are disabled so that nested interrupts cannot occur.

ISRs have the option of completely handling the interrupt, or returning a system interrupt level. In the latter case, an event is generated, notifying any device drivers that had previously requested notification for that interrupt level using the *InterruptInitialize* OS call. The driver can then process the interrupt as a user-level thread (called an Interrupt Service Thread or IST).

During execution of the OEM-specific platform initialization function (*OEMInit*), two critical steps occur to set up interrupts. First of all, the programmable interrupt controller is configured to generate the appropriate interrupt vector for each of its hardware interrupt inputs. Next, all interrupts are routed through a platform-specific interrupt dispatcher during execution of the platform's *OEMInit* function. To do this, *HookInterrupt* is called once for each interrupt vector. The common platform dispatcher, ISR, returns the system interrupt level associated with the detected hardware interrupt.

Although it is important to understand how interrupts are handled in WinCE, you shouldn't have to use *HookInterrupt* or *InterruptInitialize*, since NDIS device drivers go through the NDIS API for setting up interrupt handlers. The NDIS data structure set up by the prior call to *NdisMRegisterMiniport* includes pointers to two interrupt-related functions provided by the driver: *HandlerInterruptHandler* and *ISRHandler*.

Early in its initialization, the NDIS driver calls *NdisMRegisterInterrupt*, at which time the NDIS Wrapper creates a new IST that calls *InterruptInitialize* and then goes to sleep waiting for an interrupt. Upon awakening, the IST will call the handler pointed to by the *ISRHandler* field of the data structure. Despite its name, *ISRHandler* is not executed at interrupt level.

The *ISRHandler* function has the following prototype:

```
VOID ISRHandler(
    OUT PBOOLEAN InterruptRecognized,
    OUT PBOOLEAN QueueDpc,
    IN PVOID Context
)
```

Usually, *ISRHandler* only disables further interrupts from the network adapter and returns values of TRUE for *InterruptRecognized* and *QueueDpc*. *InterruptRecognized* can be used to handle shared interrupts, in which case a FALSE value would indicate that the generated interrupt needs to be handled by another driver. *QueueDPC* will result in a "delayed procedure call" to *HandlerInterruptHandler* by the NDIS Wrapper.

There are a couple of caveats to interrupt handling for WinCE NDIS device drivers. First, take a peek at your platform's *OEMInit* function to make sure that the hardware interrupt which your device will use is associated with a system interrupt vector in the platform interrupt dispatcher. The other thing to watch out for is that some early versions of WinCE required that the desired interrupt for the network controller be mapped to the system interrupt level indicated by the SYSINTR_NETWORK macro (which was then located in the NKINTR.H file). This was because the NDIS wrapper would use SYSINTR_NETWORK when it called *InterruptInitialize*, regardless of what interrupt level was passed to *NdisMRegisterInterrupt*.

Figure 3 summarizes the chain of events in the processing of a typical NDIS device driver interrupt.

Memory Management for Win CE NDIS Device Drivers

So now we know how WinCE NDIS device drivers are loaded and how they handle interrupts. The next thing to understand is how memory is managed in WinCE.

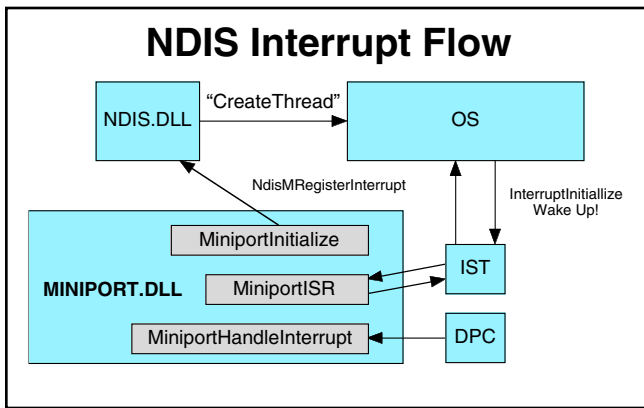


Figure 3

Regardless of the platform, WinCE runs under a flat, 32-bit address space. Applications run in private 32 Mbyte regions, implemented using virtual addressing. The OS “user address space” is the bottom 2 Gbyte (8000 0000h); physical memory is mapped starting at the top of this address space. Win32 API calls all use virtual address which exist in the user address space.

DMA and WinCE

Device drivers often use Direct Memory Access (DMA) to transfer data to and from the device. In DMA, some agent besides the CPU moves data. This agent might reside in the device itself (Master DMA), or it might be a dedicated DMA controller (Slave DMA). Regardless of which type of DMA is being used, the DMA agent must know the physical address of the data being transferred. It doesn’t know about the virtual addresses used in the Win32 API.

In the Slave DMA case, it will be necessary to program the DMA controller directly from the device driver. The NDIS Wrapper for other OSs typically provides a layer of abstraction so that this is not necessary, but this feature is not present in WinCE. There is a good example of how the DMA controller programming is accomplished in the NSC IrDA miniport example driver in the WinCE DDK.

In both kinds of DMA, device drivers typically need to be able to do one of two things to handle the translation from virtual memory to physical memory. One is to inquire the OS about the physical addresses of each “chunk” of a buffer’s data, since the buffer may not be contiguous in physical memory. In this case, it is also necessary to lock the memory so that the physical address does not change after the address has been determined. Depending on the DMA engine architecture, it may be possible to program several physical buffer addresses at one time. The other option to this “scatter-gather” approach is to allocate a buffer of memory that is specially flagged to be contiguous in physical memory.

Although NDIS API provides functions to address both of these scenarios, this subset of NDIS functionality is yet again not present in WinCE. Specific examples of these omissions include:

- *NdisMAllocateSharedMemory*
- NDIS_MEMORY_CONTIGUOUS flag of *NdisAllocateMemory*
- *NdisMStartBufferPhysicalMapping*

Using CONFIG.BIB to Reserve Buffers for DMA

To get around these deficiencies, it is often necessary in WinCE to pre-allocate memory for the network buffers using the CONFIG.BIB file that will be described below. This memory might exist on the network controller itself (memory-mapped RAM), or in system DRAM

(shared memory). In the latter case, you must “slice off” the required memory from the available system DRAM.

CONFIG.BIB is a platform-specific configuration file that indicates where the various OS modules will be located in memory. It is used by the MAKEIMG tool during creating of the OS binary image.

When memory-mapped RAM is being used, one simple change to CONFIG.BIB is required. To the MEMORY section of the file, add a line similar to this:

```
PCNETBUF      800C0000      00040000 RESERVED
```

The “PCNETBUF” label is not used by the OS. 800C0000h is the virtual address corresponding to the physical address where your memory is mapped. Remember: the CPU physical address space is mapped linearly at the top of the OS user address space, so a virtual address of 800C0000h corresponds to a physical address of C0000h. The next value (40000h) is the size of the buffer, and RESERVED keeps the MAKEIMG tool from allocating this address space for an OS module.

When shared memory is being used, we must modify CONFIG.BIB to reduce the amount of system RAM available for the OS, as well as to reserve the memory for the buffer. Again, we will be modifying the MEMORY section of CONFIG.BIB. In its original state, there should be a line such as:

```
RAM 80800000 00800000 RAM
```

This indicates 8 Mbyte of user DRAM available starting at virtual address 80800000h (physical address 800000h). To reserve 256 Kbyte of space for the buffer at the top of this region, we will change the above line to:

```
RAM 80800000 007C0000 RAM
```

```
PCNETBUF 80FC0000 00040000 RESERVED
```

You can see that we reduced the size of the “RAM” area, while creating a new area for the network buffer in its place.

The next question you should have is: how do you access memory in the CONFIG.BIB file? An important thing to remember is that you cannot use the virtual addresses that are in the CONFIG.BIB file — they are outside of the OS user address space. Instead, we must first allocate enough virtual address space for the buffer in the user address space, and then map that virtual buffer to the physical address that we have reserved in CONFIG.BIB. The code to perform this action will look something like this:

```
g_pvDmaVirtualBase = VirtualAlloc(0,BUFFER_LEN,
MEM_RESERVE,
```

```
PAGE_NOACCESS);
```

```
nTmpVal = VirtualCopy(g_pvDmaVirtualBase,
```

```
(LPVOID)(DMA_BUFFER_BASE_PA | 0x80000000),
DMA_BUFFER_LEN,
```

```
PAGE_READWRITE | PAGE_NOCACHE);
```

VirtualAlloc and VirtualCopy are Win32 API calls. As used above, VirtualAlloc will allocate a buffer in the user address space without committing it to any particular physical address. VirtualCopy will then map the buffer to its desired location. DMA_BUFFER_PA and DMA_BUFFER_LEN will typically be macros defined in a

WinCE-specific header file. Since the driver will be unaware of changes to CONFIG.BIB, there is some danger that someone will try to modify CONFIG.BIB without changing the header file of vice versa. Both files should be heavily commented to reflect this fact.

If this method of allocating and using network buffers seems like an ugly hack, it is. It doesn't allow dynamic resizing of the network buffer, it's bad for target devices that allow DRAM upgrades, and high-performance bus-mastering devices are performance-crippled since they will have to do an extra copy to the reserved buffer before data transfer can occur. Microsoft suggests using the LockPages kernel function as an alternative solution, but as of this writing I have yet to see any example code illustrating the use of these functions. If you'd like to use it, there is minimal documentation in wcecdk.chm (for WinCE 2.1x) or wcepb.chm (for WinCE 3.0).

Building and Installing the Driver

So now you understand some of the key concepts required to write your driver. The next steps are to figure out how to build the driver, and then install it into the target device. Although these steps vary depending on whether you are writing a driver for a built-in device (in which case the driver will be integrated in the OS image) or an add-on device, a good understanding of both situations will be helpful regardless of which one you choose.

Building the OS Image

First, let's take a quick look at the steps involved in building a WinCE OS image. The WinCE source tree is primarily divided into a project tree and a platform-specific tree. The project tree is further subdivided into a tree common across all platforms (called COMMON), and a tree specific to the project. The objects in the project tree are built as OS modules and components (a module contains several components), while the objects in the platform tree are usually built as individual files.

There are five main steps to build all of the required objects in the three main trees, and merge them into an OS image.

1. Compile project driver files using the BUILD tool BUILD starts in the root directory and recursively goes through each source directory, building each in turn. The SOURCES file is used to control where the compiled objects are copied to when they are built.
2. Build OS **modules** and **components** using the SYSGEN tool. The objects built by SYSGEN will usually end up in a directory called "CESYSGEN" under the current project directory. To control which modules/components are built when SYSGEN is run, you can modify the CESYSGEN.BAT file in the project tree.
3. Build platform-specific files using the BUILD tool.
4. Copy all of the required binaries to a "release" directory using the BUILDREL tool.
5. Make the OS image with MAKEIMG tool.

MAKEIMG merges the registry and database files from each of the three source trees, and uses the .BIB files from each tree to form the OS binary image.

As you can see, it is possible to compile a WinCE NDIS driver as an OS component or as an individual object file. The WinCE DDK provides examples of both, and for a built-in device it is simply a matter of choice as to which way you want to go. Obviously, it doesn't make sense for the driver for an add-on device to be built as an OS component. The drivers in the PUBLIC\COMMON\OAK\DRIVERS\NETCARD directories are examples of drivers built as OS components. The drivers in PLATFORM\CEPC\DRIVERS\IRMAC\STANDARD are examples of drivers built as individual files.

When built as an OS component, the driver is first compiled into a .LIB (static library) file using the BUILD tool. SYSGEN then links

the .LIB file into the .DLL file which is used at run-time. I'm not sure why Microsoft chose to do it like this, and it is probably possible to modify the SOURCES file so that the DLL file is created directly. However, if you want to do things the easy way, just use one of the example drivers as a template and go from there.

Files used in Building the Driver

To build your driver, a few files besides the traditional source code and header files are required. When the BUILD tool is run, it first parses the SOURCES file in the driver directory. The SOURCES file is used to control many aspects of the build, such as what type of target file will be output, where the target file will go, and what external libraries are required. The SOURCES file is similar to that used in other Microsoft OS DDKs, and can use IFDEFs to distinguish between a WinCE build and a WinNT build. Some of the variables set in the SOURCES file are shown in the table below.

After parsing the SOURCES file, BUILD then runs NMAKE to do the actual driver build. NMAKE requires the file MAKEFILE in the driver directory. However, like all Microsoft OS DDKs, MAKEFILE only contains a single line of source code that includes the "real" makefile for the current build environment.

Finally, an export file (sometimes called a .DEF file) is required. This file is used to specify what functions of the driver are actually exported to the OS. Your WinCE NDIS driver should only need to export a single function: *DriverEntry* as discussed above. All other functions in the driver will be referenced indirectly through the data structure passed to the NDIS wrapper with the *NdisMRegisterMiniport* function discussed earlier. For add-on devices, you will probably want to export an *InstallDriver* function and a PCMCIA detection function as discussed below.

Installing the Driver

Once the driver is built, it must be installed into the target device. As mentioned above, how this happens for an add-on device will be very different from a built-in device, so these two cases will be discussed separately.

To install the driver for a built-in device, there are two main steps required. The first is to modify the appropriate registry files so that the driver is loaded by the OS and bound to the appropriate protocol drivers. Thorough discussion of the registry entries needed for a WinCE NDIS driver is beyond the scope of this paper, but they are well documented in the WinCE DDK documentation. Typically, you will modify either PROJECT.REG or PLATFORM.REG depending on whether you are building the driver as an OS component or a platform object respectively. These registry files are both merged with COMMON.REG when the MAKEIMG tool is run.

The next step is to merge the device driver DLL files into the OS file system. The various .BIB files control which files go into the filesystem. A line such as:

```
pcnet.dll ${_FLATRELEASEDIR}\pcnet.dll NK SH
```

...must be added to PLATFORM.BIB or PROJECT.BIB as appropriate. The first field is the name of the destination file in the OS file system. The second field is the source file to be copied in the OS image. NK corresponds to a memory region (from CONFIG.BIB) into which the file will be placed, while S and H are file attributes for "system file" and "hidden file", respectively.

For an add-on device, the same two steps of modifying the registry and transferring the driver files to the OS file system are required. However, the method for accomplishing these steps is different. Transferring the driver files to the OS can be done through the RAPI/CE Services (now ActiveSync) interface. Typically, you will

want to provide a one-step program that will run on a host machine to transfer the files to the WinCE target device, rather than relying on the user to properly use the mobile devices functionality of Windows.

You then need some user-friendly way to modify the registry. The best way to do this is to provide an InstallDriver function in the DLL. The *InstallDriver* function can be called in one of two ways. First, if a new PCMCIA device is detected, no matching entries are found in the appropriate section of the registry (as discussed earlier), and no detection functions return positively, then the user will be prompted for the name of a .DLL file. That DLL file's "InstallDriver" function will then be executed. Of course, this means that you must export the *InstallDriver* function in the driver's .DEF file.

The dialog that prompts the user for a DLL file might be a little intimidating, so you might want to provide a very simple SETUP.EXE application that simply calls the driver's *InstallDriver* function.

There are several examples of InstallDriver functions (including how to dynamically modify the registry) in the WinCE DDK.

Porting an Existing NDIS Device Driver

Although you now have most of the information that you need to write a WinCE NDIS device driver, I want to briefly go over the

steps that those of you who are porting an NDIS driver from another OS will need to follow. These steps are:

1. Figure out the architecture for the memory buffers. Modify CONFIG.BIB to map the buffer into the address space, or figure out how you will use the LockPages function calls to get the physical addresses of the buffers you need.
2. Replace the functions that were omitted in the WinCE NDIS Wrapper. Particularly look out for functions which program the DMA controller, allocate memory, or map memory.
3. Modify the SOURCES file to reflect the target file type and destination directory, using one of the example drivers from the DDK as a reference.
4. Establish your installation procedure. This includes writing host-side and target-side installation applications (for add-on devices), or modifying .BIB and .REG files (for built-in devices).

A couple of "gotchas" when porting your driver:

- Data structures in NDIS.H are not portable. Only use NDIS API functions to access the contents of these structures.
- Some API calls were mistakenly exported into the WinCE NDIS Wrapper even though the functionality was omitted. This means that although your driver might compile with no problems, some of the functions will do nothing. If you are seeing suspicious

Variable Name	Function
TARGETNAME	Sets the base name of the driver being built.
TARGETTYPE	Controls whether an .EXE, .DLL, or .LIB file being built. OS components are typically built as a LIBRARY, and then linked into a DLL when the SYSGEN tool is run. Drivers written as platform files as opposed to OS components usually have this variable set to DYNLINK, resulting in a DLL being created when BUILD is run as opposed to later.
RELEASETYPE (optional)	Controls the directory to which the driver file will be output. If developing a stand-alone driver that the end-user or OEM will copy to the target after the OS is built, LOCAL is a good choice. If no value is specified, the driver file will go to the \$(_PUBLICROOT)OAK directory.
C_DEFINES (optional)	Controls compiler defines
DEFFILE (optional)	Specifies the name of the "source" export file. If omitted, this value will default to TARGETNAME.def. Under normal conditions, the export file specified with this macro will be located in the same directory as the source.
TARGETDEFNAME (optional)	Specifies the "destination" export file name. This is the filename where the "source" export file will be copied. The directory it will be copied to depends on the TARGETTYPE.
WINCETARGETFILE0 (optional)	If defined, should be the directory and filename indicated by TARGETTYPE and TARGETDEFNAME. Otherwise, the export file will not be copied to the target directory. Usually, this will be defined only if TARGETTYPE is LIBRARY, indicating that the library will be linked into a component DLL when SYSGEN is run.
SOURCES	Indicates the source files required for the driver.
INCLUDES	Indicates the directories that will be searched when a source file uses the #include directive.
DLENTY (optional)	Specifies the entry function when TARGETTYPE is DYNLINK. This function called when the DLL is loaded or unloaded. If undefined, DIIMainCRTStartup will be used as the entry point.
TARGETLIBS	Specifies static library files which are used by the driver. Normally used only when the TARGETTYPE is DYNLINK.

behavior, check and re-check that the NDIS API calls you are using are supported in the version of WinCE that you programming for.

Conclusion

Hopefully, from this point you will be able to look at the Windows 2000 DDK documentation for NDIS, and when you write your NDIS driver for WinCE the steps will be a little more clear. The sample NDIS drivers that Microsoft provides with the WinCE DDK are an essential reference for your development. If you are having problems, look at the example drivers and try to figure out how they differ from yours. You can also compare these drivers to those in the DDKs for other operating systems to understand some of the intricacies of WinCE. It will also be worth your while to take a half-day

or so and try develop a thorough understanding of the WinCE build process, including analysis of the systemwide MAKEFILE.DEF file and project SYSGEN.BAT files. This will be valuable in debugging build-level issues that arise. Happy networking!

Michael Migdolis is an engineer
in the EPD Systems Software Engineering
Department of Advanced Micro
located in Austin, Texas.