Powered by Microsoft® Windows® CE

MICROSOFT PROFESSIONAL EDITIONS

Microsoft® Press

The ultimate reference and toolkit for Windows CE

Microsoft® Windows® CE Device Driver Kit

Microsoft®

Windows® CE Device Driver Kit

Microsoft® Press

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Cataloging-in-Publication Data
Microsoft Windows CE Developer's Kit / Microsoft Corporation.
p. cm.
ISBN 0-7356-0619-6
1. Microsoft Windows (Computer file) 2. Operating systems (Computers) I. Microsoft Corporation.
QA76.76.O63M74515 1999
005.4'469--dc21 99-24745
CIP

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 MLML 4 3 2 1 0 9

Distributed in Canada by ITP Nelson, a division of Thomson Canada Limited.

A CIP catalogue record for this book is available from the British Library.

Acquisitions Editor: Ben Ryan
Project Editor: Alice Turner

Part No. 097-0002205

Contents

Preface

The *Microsoft® Windows® CE Developer's Kit* provides all the information you need to write applications for devices based on the Microsoft Windows CE operating system. The kit includes the following four books:

- *Microsoft® Windows® CE Programmer's Guide*

  Introduces the architecture of the Windows CE operating system.

  Explains the low-level details of creating a Windows CE–based application, including handling processes and threads, managing memory and power, accessing the object store, and modifying the registry.

  Provides information on connecting a Windows CE–based device to a desktop computer, synchronizing data between a device and desktop, and transferring files.

  Provides information on using Unicode and localizing Windows CE–based applications.

- *Microsoft® Windows® CE User Interface Services Guide*

  Describes all tasks associated with creating a user interface (UI) for a Windows CE–based device, including how to create windows and dialog boxes, how to handle messages, and how to add menus, controls, and other resources to a UI.

  Discusses how to handle various user input methods (IMs) such as keyboards and touch screens.

- *Microsoft® Windows® CE Communications Guide*

  Provides basic instructions for implementing communications support on a Windows CE–based device, including how to handle infrared connections, develop telephony applications, implement Remote Access Service (RAS) functionality into an application, handle networking and security issues, work with Windows Sockets, and establish an Internet connection.

- *Microsoft® Windows® CE Device Driver Kit*

  Provides procedures for writing device drivers for Windows CE–based devices.

  Explains how to create native and stream interface drivers as well as how to implement universal serial bus (USB) and network driver interface specification (NDIS) drivers.

The CD that accompanies the books includes online versions of the books plus the following content.

Content	Description
Windows CE API	Shows the interfaces, functions, structures, messages, and other application programming interface (API) elements for Windows CE.
Device Driver Kit API	Shows the interfaces, functions, structures, messages, and other API elements needed to create device drivers for Windows CE.
Microsoft Foundation Class (MFC) Library for Windows CE	Shows the classes, global functions, global variables, and macros needed to create full-featured Windows CE–based applications.
Active Template Library (ATL) for Windows CE	Shows the classes, macros, and global functions needed to develop small, fast Microsoft® ActiveX® controls for platforms that run Windows CE.
Mobile Channels	Demonstrates how to use Active Server Pages (ASP) and Channel Definition Format technology to enable offline Web site browsing on a Windows CE–based device.
Writing applications for a Palm-size PC	Demonstrates how to work with the Palm-size PC shell, handle memory and power, programmatically access Palm-size PC navigation controls, and design the UI for applications running on a Palm-size PC.
Writing applications for a Handheld PC	Demonstrates how to work with the Handheld PC (H/PC) shell, handle memory and power, and synchronize data between an H/PC and a desktop computer.
Writing applications for an Auto PC	Demonstrates how to implement speech, control the audio system, interact with a vehicle computer, communicate with a Global Positioning System (GPS) device, and design an effective UI for an Auto PC application.

This book, the *Microsoft Windows CE Device Driver Kit*, contains the following chapters:

Introduction to the Windows CE Device Driver Kit

This chapter provides an overview of the various device driver models used in Windows CE.

Developing Native Device Drivers

This chapter provides information about native drivers that have special-purpose interfaces to other Windows CE components.

Developing Stream Interface Device Drivers

This chapter provides information about developing device drivers for generic devices.

Audio Drivers

This chapter provides information about audio compression manager drivers and waveform audio drivers in Windows CE.

Printer Drivers

This chapter provides information about the printing model used in Windows CE and how to write device drivers for printers.

Display Drivers

This chapter provides information about the display driver model used in Windows CE and how to write device drivers for display devices.

Universal Serial Bus Drivers

This chapter provides information about USB support in Windows CE, a brief overview of the USB architecture, and information about writing device drivers for a USB device.

NDIS Network Drivers

This chapter provides information about the network driver interface specification (NDIS) driver model in Windows CE and how to write device drivers for networking devices.

Block Device Drivers

This chapter provides information about block device drivers in Windows CE and how to write device drivers for block devices, such as linear flash memory devices.

Document Conventions

The following table shows the typographical conventions used throughout this book.

Convention	Description
`monospace`	Indicates source code, structure syntax, examples, user input, and application output. For example, `ptbl->SortTable(pSort, TBL_BATCH);`
Bold	Indicates an interface, method, function, structure, macro, or other keyword in Windows CE, the Microsoft Windows operating system, C, or C++. For example, **CommandBar_Height** is a function. Within discussions of syntax, bold type indicates that text must be entered exactly as shown.
Italic	Indicates placeholders, most often method or function parameters; these placeholders stand for information that must be supplied by the implementation or the user. For example, *lpButtons* is a function parameter. Also indicates new terms defined in the glossary.
UPPERCASE	Indicates flags, return values, messages, and properties. For example, WSAEFAULT is a Windows Sockets error value, MF_CHECKED is a flag, and TB_ADDBUTTONS is a message. In addition, uppercase letters indicate segment names, registers, and terms used at the operating-system command level.
( )	Indicate one or more parameters that you pass to a function, in syntax.

CHAPTER 1

Introduction to the Windows CE Device Driver Kit

Like other operating systems, Windows CE implements software called *device drivers*, whose purpose is to interface with, or drive, built-in and peripheral hardware devices. A device driver links an operating system and a device, making it possible for the operating system to recognize the device and to present the device's services to applications.

Windows CE supports a wide range of device drivers that you can customize for various Windows CE–based platforms. Windows CE also provides several models for driver development, including driver models from other operating systems. Because of this diversity of driver models, Windows CE accommodates most devices, either as a built-in device or as a peripheral device.

The *Microsoft Windows CE Device Driver Kit* supplies documentation enabling you to create device drivers for Windows CE. Currently, Windows CE offers four device driver models, two that are unique to Windows CE and two that are external models adapted from other operating systems. The two Windows CE–based models are the native device driver and the stream interface driver. The two external models are designed for universal serial bus (USB) drivers and network driver interface specification (NDIS) drivers.

The various driver models are differentiated only by the software interface that they support and not by the devices that they serve. The driver model determines the software interface that a specified driver exports. Independent of the driver model, a device driver can be either monolithic or layered, meaning that it can implement its software interface directly in terms of actions on the device or it can separate the implementation of the software interface and the actions on the device into two layers. Many of the sample drivers provided by Microsoft use this layered organization because it reduces the amount of code that developers must write when porting the sample drivers to new devices. Device drivers can access their devices directly if the devices are mapped into system memory, or they may need to use the services of lower-level device drivers in order to access their devices.

For example, device drivers for PC Card devices need to use the services of the PC Card socket driver to access PC Card devices. Finally, device drivers may be interrupt-driven, may be polled, or may not require status updates from their devices.

As their name suggests, native device drivers serve devices that are integral to a Windows CE–based platform. Native device drivers are designed specifically for low-level, built-in hardware, such as keyboards, screens, and PC Card sockets. Because native device drivers generally have a strong connection to Windows CE–based platforms and each type of native device driver has a precise, specialized purpose, Microsoft provides support for native device drivers in the form of custom interfaces. This means that the majority of developers do not need to write native device drivers. The exceptions are original equipment manufacturers (OEMs) who customize Windows CE for new platforms. They can either create their own native device drivers or port Microsoft sample native device drivers to their Windows CE–based platform. Native device drivers are always loaded when the Windows CE–based platform starts up.

In contrast to native device drivers, which have custom interfaces, stream interface drivers are a generic type of device driver. Stream interface drivers are user-level dynamic-link libraries (DLLs) that implement a fixed set of functions—the stream interface functions—enabling an application to interact with a device through special files in the file system. Stream interface drivers support almost any kind of peripheral device that can be attached to a Windows CE–based platform. For example, developers have designed stream interface drivers to support a variety of peripherals, including pagers, printers, modems, bar code scanners, and Global Positioning System (GPS) receivers.

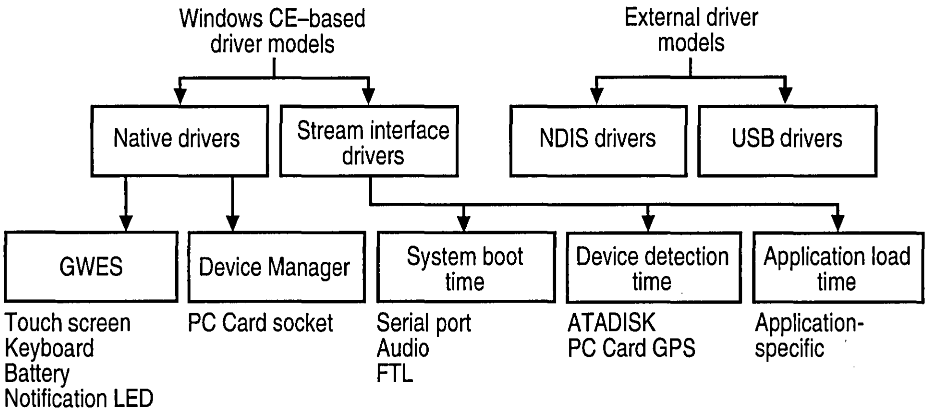
The stream interface functions enable applications to interact with a device through special files in the file system. OEMs can also create stream interface drivers to serve devices that are designed as part of a Windows CE–based platform when Microsoft has not defined a specific interface for a particular type of device. For example, a Windows CE–based point-of-sale system might have a bar code scanner built into the hardware of the system, but because Microsoft has not yet defined a custom interface for bar code scanners, the device driver for the scanner would be a stream interface driver, rather than a native device driver.

Some common types of built-in devices, such as serial ports, use stream interface drivers because the functionality of the devices is well suited to the structure of a stream interface driver. For this reason OEMs sometimes write a few stream interface drivers. Peripheral manufacturers are responsible for providing stream interface device drivers so that their peripherals can be used with Windows CE.

A universal serial bus driver (USBD) connects USB-compliant devices to Windows CE. Unlike stream interface drivers, however, USBDs are not required to export any particular set of functions to applications. Depending on the device, a USBD can export the stream interface functions, export a custom set of functions, or use existing Windows CE APIs to expose the functionality of the device. Using existing Windows CE APIs is appropriate for a pointing device or a mass-storage device, neither of which applications need to use directly. Windows CE has existing mechanisms, the input-event system and the block device driver API, respectively, for managing these types of resources.

NDIS drivers are adapted from the Microsoft® Windows NT® operating system. The NDIS driver is a driver model that enables networking protocols, such as TCP/IP and the Infrared Data Association (IrDA) protocol, to be completely independent of the implementation details of device drivers for network interface cards (NICs).

The following illustration shows the current collection of driver models and device drivers.



The remainder of this documentation explains in detail how to implement the device driver models.

The Device Driver Kit is packaged with two different products, the Microsoft® Windows® CE Platform Builder, which contains source code and libraries for OEMs who build custom Windows CE–based platforms, and the Windows CE Platform SDK, which contains the cross-compiler tools and libraries necessary to build software for Windows CE–based platforms. In addition, the Platform Builder includes source code for a number of sample device drivers.

CHAPTER 2

Developing Native Device Drivers

To port Windows CE to a target platform, you must provide device drivers for the devices built into your platform. Some types of devices—such as keyboards, displays, and PC Card sockets—have a custom interface to the operating system. The drivers for these types of devices are called native device drivers because the interfaces they use are specific to Windows CE.

In general, native device drivers are of interest only to original equipment manufacturers (OEMs) who build Windows CE–based platforms. Independent hardware vendors (IHVs) who develop drivers for add-on hardware have no need to design or customize native device drivers. Therefore, the following sections regarding native device drivers are directed primarily to OEMs.

Microsoft defines custom interfaces for each type of native device driver. However, although each type of native device driver has a custom interface, native device drivers present a standard set of functionality for all devices of a particular class. This enables the Windows CE operating system to treat all instances of a particular device class alike, despite any physical differences. For example, many Windows CE–based platforms use some type of LCD panel as their display. However, there are a wide variety of these panels on the market that have different operating characteristics, such as resolution, bit depth, memory interleaving, and so on. By making all display drivers conform to the same interface, Windows CE can treat all display devices the same, regardless of the physical differences between the devices themselves.

Typically, OEMs link native device drivers with the Graphics, Windowing, and Events Subsystem (GWES) module when they build custom Windows CE–based platforms. However, there are exceptions; not all native device drivers are linked with GWES during the build process. For example, users install and load some types of native device driver, such as display drivers and printers.

For a Handheld PC (H/PC), sample native device drivers are available for the various device classes built into the platform. These device classes are the following:

- Display
- Battery
- Keyboard
- Touch screen
- Notification LED
- PC Card socket

If your target platform contains a set of devices different from those on the H/PC, you need to create your own native device drivers for the devices. However, if your platform includes devices similar to those on the H/PC, consider porting the sample native device drivers—which the Windows CE Platform Builder provides—to your platform, rather than developing your own native device drivers. By porting the tested device drivers, you can save time and avoid bugs. Even so, it is not mandatory to use the sample code provided with the Platform Builder for the native device driver or for any other driver model. The Platform Builder supplies driver samples solely as a convenience to help you develop your drivers rapidly.

The following sections provide information about native device drivers to help you use and modify the driver models supplied in the Platform Builder. These sections discuss the following:

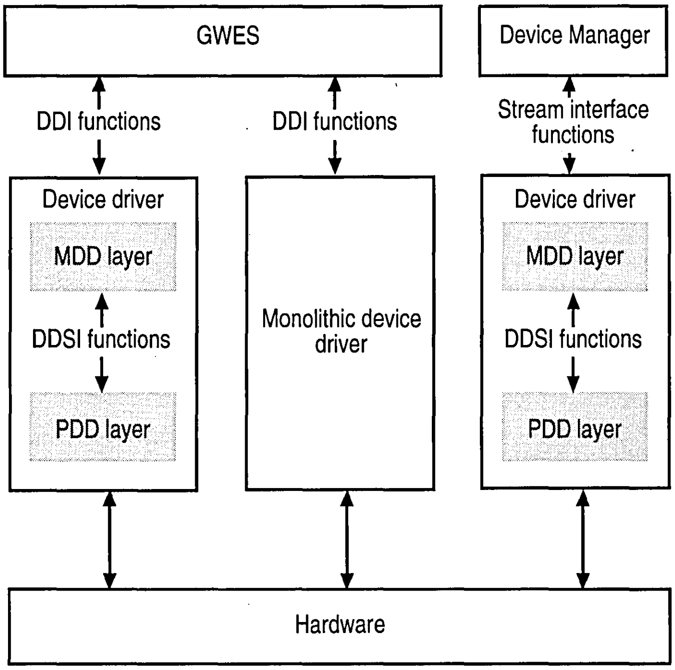
- System architecture for native device drivers
- Restrictions on what native device drivers can do and what system APIs they can call
- Handling of interrupt events in a native device driver
- Porting of sample native device drivers to your platform

Note The directory and paths listed are relative to the root directory of the Platform Builder installation.

System Architecture for Native Device Drivers

The sample device drivers included with the Platform Builder come in two types: *monolithic* and *layered*. A monolithic driver, as the name implies, is based on a single piece of code that exposes the functionality of the hardware device directly to the operating system. In contrast to the monolithic driver, the layered driver consists of two customized layers: the upper layer is the model device driver (MDD), and the lower layer is the platform-dependent driver (PDD). Most of the sample device drivers are configured as layered, rather than monolithic.

The following illustration shows the integration of monolithic and layered drivers within the Windows CE operating system.



Microsoft provides the MDD for a layered driver. The MDD is common to all platforms and functions, both as source code and as a library. It performs the following tasks:

- Links to the PDD layer and defines the functions it expects to call in that layer
- Exposes different sets of functions to the operating system
- Handles complex tasks, such as interrupt processing
- Communicates with the GWES module and with the kernel

Each MDD also handles a specific set of devices, such as audio hardware or touch screens. In general, the MDD requires no changes. If you choose to modify the MDD, be aware that Microsoft does not test, warrant, or support custom MDDs. You are responsible for all further MDD maintenance if Microsoft supplies an updated MDD in order to fix bugs or to support later versions of Windows CE. In addition, if you revise the MDD you must provide support to any IHVs who use those changes.

Unlike the MDD layer, the PDD layer, which interfaces with both the MDD and the hardware, is meant to be tailored to your target platform. A PDD consists of hardware-specific functions that correspond to an MDD. There is no direct one-to-one relationship between the functions in a PDD and the corresponding MDD; the MDD functions implement discrete tasks that the MDD uses to achieve its goals. Because the PDD is hardware-dependent, you must create a customized PDD and port it to your platform hardware. To assist you, Microsoft provides several sample PDD layers for various built-in devices.

You can forego the MDD and PDD layers by implementing your device driver as a monolithic driver. For example, if performance is a critical factor, a monolithic driver might be a better choice than a layered driver because a monolithic driver avoids the overhead associated with the function calls that take place between the MDD and PDD layers. You might also choose to implement a monolithic driver if the capabilities of the device in question are well matched to the tasks that the functions in the MDD layer perform. In such a case, implementing a monolithic driver might be simpler and more efficient than implementing a layered driver. However, regardless of whether you implement a monolithic driver or a layered driver, you can base your implementation on the source code for any of the sample layered drivers.

As shown in the previous illustration, the device driver interface (DDI) is a set of functions implemented in the MDD and called by the GWES module; the device driver service-provider interface (DDSI) is a set of functions implemented in the PDD and called by the MDD. Use DDI functions for monolithic drivers and DDSI functions for layered drivers.

Finally, some of the sample device drivers are implemented as stream interface drivers, which means that the drivers use the stream interface as their DDI. In this case, you do not need to link such drivers with the GWES module. They exist as ordinary DLLs and are loaded as needed. The audio driver is an example of a device driver using the stream interface model. Because these drivers have an MDD and PDD that you can use as a basis for your development efforts, they are included here in the native device driver section, rather than in "Developing Stream Interface Device Drivers."

Restrictions on Native Device Drivers

Device drivers are user-level processes, which means that the driver code can call Microsoft® Win32® APIs and access any resources available to user-level processes. For the most part, though, device drivers need only a limited number of simple APIs, such as memory allocators. Drivers do occasionally perform more complex tasks, including creating threads or windows. For example, on many Windows CE–based platforms, the battery driver presents a dialog box to users when the batteries drop below a threshold voltage, thus notifying users to replace the batteries.

The only time that a driver cannot call Win32 APIs is when the driver processes a notification that the device is shutting down. In this situation, the device driver must not perform any operations that might cause a context switch. For example, if the driver attempts to open a file during this time, the kernel might need to access the file system, swap pages out of memory, and use other system resources that might depend on separate execution contexts. To avoid these problems, restrict the device driver to the following actions when shutting down:

- Saving the device state in preallocated storage
- Issuing any commands to the device pertinent to shutting down
- Setting a flag within the driver to note that it has been shut down

When power returns, the driver can restore the device state from the saved data.

The driver can also use the **SetInterruptEvent** function to generate an artificial interrupt event. By calling **SetInterruptEvent**, a driver can force release of the interrupt thread to continue any necessary processing. For examples of using **SetInterruptEvent**, see the **HWPowerOn** and **HWGetIntrType** functions in the P2io.c file. Drivers should call these functions only during power cycling.

Interrupt Handling in Native Device Drivers

Like many other computer architectures, Windows CE–based platforms use interrupts to signal the operating system when devices need servicing by their drivers. For example, when a user presses a key or taps a screen, the keyboard hardware or the touch screen generates an interrupt.

However, Windows CE does not support nested interrupts, meaning that interrupts cannot occur simultaneously. Processing of one interrupt must finish before another begins. Although this limits the real-time scheduling capabilities of Windows CE, it vastly simplifies the process of writing interrupt handlers.

Windows CE balances performance and ease of implementation by breaking interrupt processing into two parts, a kernel-mode part and a user-mode part. The kernel-mode part is called the interrupt service routine (ISR), and the user-mode part is called the interrupt service thread (IST). The ISR is the part that cannot be nested; therefore, keep your ISRs short and fast so that interrupts are disabled as little as possible. The ISR resides in the OEM adaptation layer (OAL) and has direct access to hardware registers. Its sole job is to determine which interrupt identifier, such as SYSINTR_SERIAL, to return to the kernel interrupt handler.

Essentially, ISRs map physical interrupts onto logical interrupts. Platform-independent interrupt identifiers are declared in the Nkintr.h header file, and platform-specific interrupt identifiers are declared in platform header files, such as Platform\Cepc\Inc\Oalintr.h.

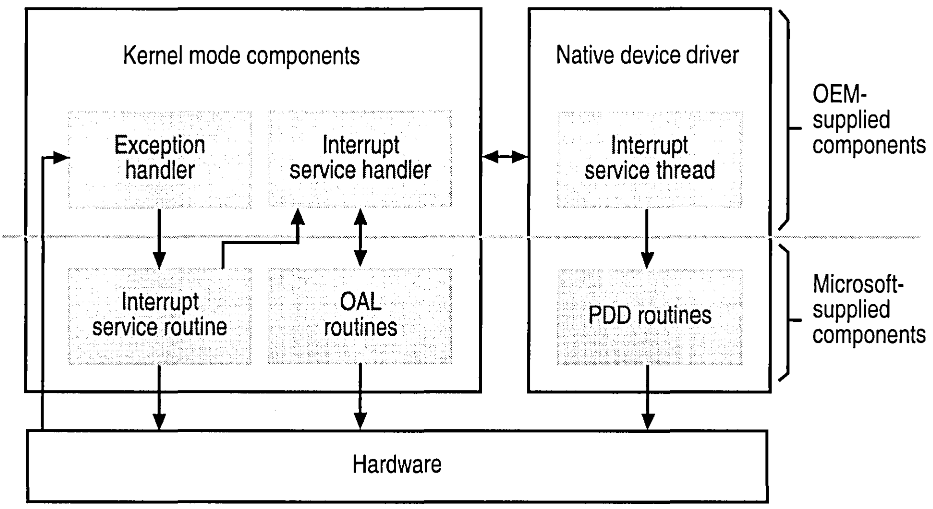
Because an ISR is small and does very little processing, the interrupt handler calls an IST to perform the bulk of processing required to handle an interrupt. The IST remains idle until it receives a signal from the **WaitForSingleObject** function that an interrupt has occurred. When an IST receives the signal, it calls subroutines in the PDD layer of the native device driver; these routines in turn access the hardware to retrieve information or to set the hardware state. The IST is a standard secondary thread that can be preempted and scheduled by the operating system. However, to enhance real-time performance, Windows CE uses the two highest thread-priority levels, THREAD_PRIORITY_HIGHEST and THREAD_PRIORITY_TIME_CRITICAL, for ISTs. These priority levels are not reserved for ISTs; any secondary thread can also have them.

On a Windows CE–based reference platform, which uses the Hitachi SH3 microprocessor, the system can start an ISR in 2 to 5 microseconds. The system can start the corresponding IST in 90 to 170 microseconds. Many actual Windows CE–based platforms have better response times than these, depending on factors such as CPU type, clock speed, bus speed, and so on. Aside from hardware performance, the IST start times vary from one interrupt event to the next due to unpredictable factors, such as the state of the processor cache and whether the IST happens to be the currently running thread.

If your platform is not meeting your real-time performance requirements, you have the following options:

- Putting more processing into the ISR. This is strongly discouraged because when an ISR is running, no other threads are scheduled and interrupts are masked. A user might experience poor device performance or missed input if an ISR takes too long.
- Adding buffering hardware that can store data relevant to several interrupt events and modifying the IST so that it gathers the data from several interrupt events in the buffering hardware. This enables the IST to process interrupts in batches, resulting in a reduction of real-time requirements.
- Using a higher clock frequency on the CPU, on the data bus, or on both.
- Adjusting the priorities of the platform ISTs.

The following illustration shows the interaction of components during the interrupt process.



The exception handler is the primary target of all interrupts. When an interrupt occurs, the kernel jumps directly to the exception handler. The exception handler then calls whichever ISR is registered to handle the current interrupt.

OEMs register their ISRs with the exception handler at startup time. First, the kernel calls the **OEMInit** function in the OAL. Next, **OEMInit** calls the **HookInterrupt** function to inform the exception handler of which ISRs correspond to individual physical interrupt lines. A few subroutines in the OAL, such as the **OEMInterruptEnable**, **OEMInterruptDisable**, and **OEMInterruptDone** functions, are also used in interrupt processing.

The interrupt handler manages the details of registering a driver for a particular interrupt and deregistering it later. It also maintains communication among the ISR, the IST, and subroutines within the OAL.

Monolithic drivers can use the Interrupt Event Handler Support interface provided by the kernel. This interface is a lightweight process-synchronization model based on standard Win32 events. The interface consists of event-related functions, such as **CreateEvent**, **SetEvent**, **ResetEvent**, and **WaitForSingleObject**, and the Windows CE kernel **InterruptDone**, **InterruptDisable**, **InterruptEnable**, and **InterruptInitialize** functions. Layered drivers use the same functions to process interrupts, but you need not use these functions to port layered drivers because the MDD layer—which does not require modification—already contains the proper calls to these functions.

The following sections discuss in greater detail how to register an interrupt handler, process it, and deregister it.

Registering an Interrupt Handler

After a native device driver is loaded, the driver creates an IST and registers it with the interrupt handler. The IST is registered for one or more logical interrupts.

If you use the existing MDD implementation for a particular driver, the MDD layer registers the driver for interrupts. If you write a monolithic driver, you need to implement code for registering the IST of the driver with the interrupt handler. To do this, use the **InterruptInitialize** function to create an event and associate it with an interrupt identifier.

Processing an Interrupt

A specific sequence of events takes place when an interrupt is processed. You should write the ISR and IST for your native device driver with the following sequence of events in mind:

1. When an interrupt occurs, the kernel jumps to the exception handler.
2. The exception handler disables all interrupts and calls the appropriate ISR for the physical interrupt line.
3. The ISR returns a logical interrupt, in the form of an interrupt identifier, to the interrupt handler.
4. The interrupt handler re-enables all interrupts, with the exception of the current interrupt, and signals the appropriate IST.
5. The IST calls PDD routines to access the hardware and finish processing the logical interrupt.
6. The IST calls the **InterruptDone** function.
7. The interrupt handler re-enables the current interrupt and calls the **OEMInterruptDone** function in the OAL.

Deregistering an Interrupt Handler

If a device driver needs to stop processing an interrupt, the driver must use the **InterruptDisable** function. When the driver calls this function, the interrupt handler removes the association between the IST and the specified logical interrupt. The interrupt handler accomplishes this by calling the **OEMInterruptDisable** function to turn off the interrupt. The driver can register for the interrupt again later, if necessary.

Porting the Sample Native Device Drivers

To port Windows CE to a target platform, you must provide drivers for built-in devices, such as keyboards, touch screens, PC Card sockets, serial ports, and audio hardware.

To assist you with the porting procedure, the following sections contain descriptions of sample MDD and PDD implementations provided with the Platform Builder. The actual samples appear under the Platform\ODO\Drivers and Platform\ODO\GWE subdirectories of your installation. The following sections also list the DDI and DDSI functions related to the sample drivers; any native device driver you write should use or implement the same DDI or DDSI functions, depending on whether you write a monolithic or a layered device driver:

- Sample Battery Driver
- Sample Display Driver
- Sample Keyboard Driver
- Sample Notification LED Driver
- Sample PC Card Socket Driver
- Sample Serial Port Driver
- Sample Touch Screen Driver
- Sample USB Host Controller Driver

Sample Battery Driver

The battery driver provides information about the power level of the battery supply of a platform. The battery driver reports main battery status and backup battery status, assuming that a platform has both. The battery driver is statically linked to GWES, so it does not exist as a dynamic-link library (DLL). The battery driver sample code is under \Platform\ODO\GWE\Battery.

The battery driver is a monolithic driver, and hence uses only DDI functions. The following list shows the DDI functions for the battery driver:

BatteryDrvrGetLevels

BatteryDrvrGetStatus

BatteryDrvrSupportsChangeNotification

Sample Display Driver

The display driver controls all writing to the built-in display hardware of the system. Typically, this is an LCD screen, but it can be any sort of display hardware that is suitable to your target platform. In Windows CE 2.0 and later, the display driver model for built-in display hardware is the same as the model for peripheral display adapter drivers.

By default, Windows CE uses built-in display hardware that cannot be removed by a user. Windows CE can also use removable peripheral display adapters, usually in the form of PC Cards. For either type, your display driver DLL must be named Ddi.dll.

Create a registry key under **HKEY_LOCAL_MACHINE\Drivers\Display** to store configuration entries for your display driver. In addition, the initialization routine of your display driver should create keys under **HKEY_LOCAL_MACHINE\Drivers\Display\Active** so that applications can determine which display devices are available.

Sample Keyboard Driver

The keyboard driver converts input from the keyboard hardware into keyboard events entered into the input system. Next, the driver generates the proper Unicode characters from these keyboard events. The Platform Builder includes a keyboard driver that is divided into blocks, which facilitates development of keyboard drivers for international keyboard layouts. This driver can be found under \Platform\ODO\Drivers\Keybd2.

The keyboard driver contains two functions, **ScanCodeToVKeyEx** and **KeybdDriverVKeyToUnicode**, that control how keyboard scan codes are translated into virtual keys and Unicode characters. These functions are based on translation tables, which can be customized for different languages. You can create your own translation tables or customize the existing tables, if necessary.

The sample code shows a design that works for the standard Windows CE–based keyboard layout. By replacing the PDD layer, developers have implemented drivers for hardware that uses software scanning, hardware scanning, and a dedicated keyboard controller. Unusual hardware or nonstandard keyboard layouts might require modification of both the MDD and PDD layers. You can change the MDD layer to build a custom Keybdmdd.lib. Source code for the MDD layer is located under \Public\Common\Oak\Drivers.

The Windows CE input system loads the keyboard driver at startup time. When the input system starts, it retrieves the name of the keyboard driver DLL from the **HKEY_LOCAL_MACHINE\Hardware\DeviceMap\KEYBD\Drivername** registry key. If no entry is found, the input system uses the default name, Keybddr.dll. It then loads the DLL and verifies that all required entry points exist.

Next, the input system calls the **KeybdDriverInitialize** function to perform a one-time initialization. In this function, the MDD saves a local copy of the input system callback function. The MDD also calls the **KeybdPdd_InitializeDriver** function, which starts a thread to handle keyboard interrupts, and then returns to the input system.

In the sample MDD, the IST of the keyboard driver is named *KeybdDriverThread*. This thread calls the **InterruptInitialize** function to register for the SYSINTR_KEYBOARD interrupt, and then waits for the system to issue SYSINTR_KEYBOARD signals. When it receives those signals, the interrupt handler calls the **KeybdPdd_GetEvent** function. The interrupt handler sends the key events returned from the PDD to the input system, which queues them for distribution to applications.

When the input system pulls the keyboard event from the queue, it invokes a callback to the driver's **KeybdDriverVKeyToUnicode** routine. The driver analyzes the specified key event and the virtual key state and generates the corresponding characters. The input system then sends the virtual key and the characters to the appropriate application.

The **KeybdDriverGetInfo** and **KeybdDriverSetMode** functions get and set information about the keyboard. When the main input thread processes a keyboard-connection event, the thread calls the **KeybdDriverGetInfo** function to get the virtual-key-to-Unicode data supplied by the driver. The thread also allocates the required memory for the virtual key state data and any extra data required by the driver. Next, it calls the **KeybdDriverInitStates** function to initialize the memory.

The following list shows the DDI functions for the keyboard driver:

KeybdDriverGetInfo

KeybdDriverInitialize

KeybdDriverInitStates

KeybdDriverPowerHandler

KeybdDriverSetMode

KeybdDriverVKeyToUnicode

KeybdEventCallback

The following list shows the DDSI functions for the keyboard driver:

KeybdPdd_DllEntry

KeybdPdd_GetEvent

KeybdPdd_InitializeDriver

KeybdPdd_PowerHandler

Sample Notification LED Driver

The notification LED driver handles all requests to turn the system notification LED on or off. These include queuing notification events, waiting until those events have elapsed, and getting and setting user-notification preferences. The sample notification LED driver code appears under \Platforms\ODO\GWE\Nleddrv.

The following list shows the DDI functions for the notification LED driver:

CeClearUserNotification

CeHandleAppNotifications

CeRunAppAtEvent

CeRunAppAtTime

CeSetUserNotification

The following list shows the DDSI functions for the notification LED driver:

NLedDriverDllEntry

NLedDriverGetDeviceInfo

NLedDriverInitialize

NLedDriverPowerDown

NLedDriverSetDevice

Sample PC Card Socket Driver

The PC Card socket driver manages any PC Card sockets on a Windows CE–based platform. The MDD layer of this driver exposes a number of functions that can be used to write stream interface drivers for individual PC Cards. These functions constitute the Card Services library. The PDD layer exposes a set of lower-level functions that the MDD layer uses. These functions constitute the Socket Services library. The sample PC Card driver code appears under \Platform\ODO\Drivers\Pcmcia.

The following list shows the DDI functions for the PC Card socket driver:

CardAccessConfigurationRegister

CardDeregisterClient

CardGetEventMask

CardGetFirstTuple

CardGetNextTuple

CardGetParsedTuple

CardGetStatus

CardGetTupleData

CardReleaseExclusive

CardReleaseIRQ

CardReleaseSocketMask

CardReleaseWindow

CardRequestConfiguration

CardRequestExclusive

CardRequestIRQ

CardRequestSocketMask

CardMapWindow	**CardRequestWindow**
CardModifyWindow	**CardResetFunction**
CardRegisterClient	**CardSetEventMask**
CardReleaseConfiguration	

The following list shows the DDSI functions for the PC Card socket driver:

PDCardGetAdapter	**PDCardResetSocket**
PDCardGetSocket	**PDCardSetAdapter**
PDCardGetWindow	**PDCardSetSocket**
PDCardInquireAdapter	**PDCardSetWindow**
PDCardInquireWindow	**PDCardWriteAttrByte**
PDCardReadAttrByte	**PDCardWriteCmnByte**
PDCardReadCmnByte	

Sample Serial Port Driver

The serial port driver handles any I/O devices that behave like serial ports, including those based on 16450 and 16550 universal asynchronous receiver-transmitter (UART) chips and those that use direct memory access (DMA). Many Windows CE–based platforms have devices of this type, including ordinary nine-pin serial ports, infrared I/O ports, and PC Card serial devices, such as modems. If multiple types of serial ports exist on a platform, you can either create several different serial drivers, one per serial port type, or create several different PDD layers and link them to a single MDD, thus creating one multipurpose serial driver. Either approach is equally acceptable. Creating separate drivers can simplify debugging and maintenance because each driver supports only one type of port. Creating a multipurpose driver, such as the sample serial port driver for the Windows CE reference platform, is more complex but gives a small memory savings.

The sample serial port MDD driver code appears under Public\Common\Oak\Drivers\Serial. The PDD code for various platforms is in Platform\CEPC\Drivers\Ser_Pdd and Platform\Odo\Drivers\Serial. There are sample serial port PDDs that support standard 16550, PC Card–based, and infrared serial ports. The sample MDD also supports linking multiple PDD layers, as was previously described, by using a function pointer array that defines the types of supported serial port. If you add serial devices with unusual properties or functionality, you can add new serial port PDDs. In addition, Microsoft provides a library of functions appropriate for supporting serial ports that use 16550-compatible UARTs under Public\Common\Oak\Drivers\Ser16550.

The **COM_Open** function in the serial port MDD does not support the *fErrorChar*, *fNull*, and *fAbortOnChar* members of the serial device control block (**DCB**) structure. You are expected to add support for this functionality in your PDD layer, if it is appropriate for your serial port hardware.

Because the functionality of serial ports maps easily to the functionality provided by standard stream interface functions, the serial port driver uses the stream interface functions as its DDI. Like other drivers that expose stream interface functions, it is loaded by the Device Manager.

Serial port devices make extensive use of I/O control codes to set and query various attributes of a serial port. For example, there are I/O control codes for setting and clearing the Data Terminal Ready (DTR) line, for purging any undelivered data, and for getting the status of a modem device. Therefore, you should support as many **IOCTL_SERIAL** I/O control codes as possible in your implementation of **COM_IOControl**.

The following list shows the DDI functions for the serial port driver:

COM_Close	**COM_PowerDown**
COM_Deinit	**COM_PowerUp**
COM_INIT	**COM_Read**
COM_IOControl	**COM_Write**
COM_Open	

The following list shows the DDSI functions for the sample serial port driver:

GetSerialObject	**HWGetIntrType**	**HWPutBytes**
HWClearBreak	**HWGetModemStatus**	**HWReset**
HWClearDTR	**HWGetRxBufferSize**	**HWSetBreak**
HWClearRTS	**HWGetRxStart**	**HWSetCommTimeouts**
HWClose	**HWGetStatus**	**HWSetDCB**
HWDeinit	**HWInit**	**HWSetDTR**
HWDisableIR	**HWOpen**	**HWSetRTS**
HWEnableIR	**HWPowerOff**	**HWTxIntrHandler**
HWGetBytes	**HWPowerOn**	**HWXmitComChar**
HWGetCommProperties	**HWPurgeComm**	

These function names are listed as they are defined in the sample serial port code. You can change this set of functions or the function names by modifying the serial object table, which is a table of **HWOBJ** type objects, in the serial port PDD.

Power Management in Serial Port Drivers

The minimum of power management that a serial port driver can perform is to put the serial port hardware into its lowest power consumption state with the driver's **HWPowerOff** function, and to turn the serial port hardware fully back on with the driver's **HWPowerOn** function. Both of these functions are implemented in the serial port driver's PDD layer. Beyond that minimal processing, a serial port driver can conserve power more effectively by keeping the port powered down unless an application has opened the serial port. Most serial port hardware can support reading the port's input lines even without supplying power to the serial. Consult the documentation for your serial port hardware to determine what parts of the serial port circuitry can be selectively powered on and off, and what parts need to be powered for various conditions of use. If there is no need for the driver to detect docking events for removeable serial port devices, the driver can go one step further and remove power from the serial port's UART chip, provided that no applications are using the port.

Automatic Detection of Docking for Serial Ports

To support automatic detection of docking events for removeable serial port hardware, serial port drivers can monitor the Data Carrier Detected (DCD) line. For any serial port which is subject to automatic detection, such as PC Card–based serial port hardware, the driver can use the **CeEventHasOccured** function to watch for the NOTIFICATION_EVENT_RS232_DETECTED message. When such a message occurs, the user has docked a serial port. Some Windows CE–based platforms that support Direct Cable Connect synchronization services starts the synchronization services when this event occurs. See Platform\Odo\Drivers\Serial\p2io.c for an example of detecting this message.

Sample Touch Screen Driver

The touch screen driver reads input from the touch screen hardware and converts it to touch events entered into the input system. The driver is also responsible for converting uncalibrated coordinates to calibrated coordinates that take into account any hardware anomalies, such as skew or nonlinearity. This calibration is generally done when the driver is first loaded, after a cold startup of a Windows CE–based platform. The sample code appears under \Platform\ODO\Drivers\Touchp.

The driver must submit points while the stylus is touching the touch screen. When the stylus is removed from the touch screen, the driver must submit at least one final event indicating that the stylus tip was removed. The calibrated coordinates must be reported to the nearest one-fourth pixel.

The following pseudocode example shows the calibration algorithm.

```
TouchPanelEnable(...);  // Start the panel sampling.
TouchPanelGetDeviceCaps(...); // Request number of calibration points.
for (i=0; i<# points; i++)
  {
    TouchPanelGetDeviceCaps(...); // Get a calibration coordinate.
    //Draw crosshair at returned coordinate.
    TouchPanelReadCalibrationPoint(...); // Get calibration data.
  }
TouchPanelSetCalibration(...); // Calculate calibration coefficients.
```

After calibration, the touch screen driver sends any pen samples generated for the touch screen to the callback function specified in the **TouchPanelEnable** function. The driver must pass calibrated points to the callback function.

The following list shows the DDI functions for the touch screen driver:

ErrorAnalysis
TouchPanelCalibrateAPoint
TouchPanelDisable
TouchPanelEnable
TouchPanelGetDeviceCaps
TouchPanelPowerHandler
TouchPanelReadCalibrationAbort
TouchPanelReadCalibrationPoint
TouchPanelSetCalibration
TouchPanelSetMode

The following list shows the DDSI functions for the touch screen driver:

DdsiTouchPanelDisable
DdsiTouchPanelEnable
DdsiTouchPanelGetDeviceCaps
DdsiTouchPanelGetPoint
DdsiTouchPanelPowerHandler
DdsiTouchPanelSetMode

Sample USB Host Controller Driver

This section describes the specific aspects of universal serial bus (USB) support in Windows CE 2.10 and later that are relevant to OEMs who want to add USB support to their Windows CE–based platforms. For a complete overview of USB support in Windows CE, including information directed to IHVs, see "Universal Serial Bus Drivers."

The primary goal of Windows CE USB support, aside from enabling IHVs to write device drivers for USB devices, is to help OEMs expand existing USB support on their platforms. Currently, USB support includes only the host side of the USB specification, which enables Windows CE to support USB peripherals. OEMs are free to add device-side support if their Windows CE–based platforms need to act as USB peripherals to other USB hosts.

OHC and UHC Host Controllers

The host controller, or adapter, is a hardware layer contained within the host computer; in this case, a Windows CE–based platform. The host controller converts data between the format used by the host computer and the format used on the bus. There are two standard host-controller designs, open host controller (OHC) and universal host controller (UHC). OEMs are responsible for providing driver software for the particular host controller hardware present on their Windows CE–based platforms. Microsoft supplies a sample host controller driver that an OEM can use as a basis for a host controller driver (HCD). For more information about USB host controllers, see "Writing an HCD Module."

HCD Nested Hub Support

Windows CE guarantees support for connecting hub devices only up to one level deep. In laboratory settings, the HCD module supplied by Microsoft has supported several layers of nested hubs; however, this feature could not be thoroughly tested and certified before the release of Windows CE 2.10. OEMs who want to support multiple layers of nested hubs are strongly encouraged to test the behavior and power consumption of peripherals thoroughly under a wide variety of hub configurations before announcing support for multiple levels of hubs in their products.

Suspending and Resuming

Windows CE supports suspending and resuming USB devices in association with the standard Windows CE power states. When Windows CE issues a POWER_DOWN notification, the HCD module suspends the USB host controller hardware and all devices. To achieve this, the MDD layer of the HCD module calls a function in the PDD layer to enable the HCD module to complete any platform-specific processing needed to suspend the host controller hardware properly. Suspending power to the host controller hardware typically causes USB devices connected to a Windows CE–based platform to enter the suspended state. However, this is not recommended for all devices; USB devices that can collect and store data while the host computer is off should not be suspended.

When the Windows CE–based platform is turned on again, Windows CE sends a POWER_UP notification to the HCD module. Next, the MDD layer of the HCD module calls a function in the PDD layer. Because the PDD layer is used, OEMs can customize the HCD module to perform any necessary platform-specific processing. Following the call to the PDD layer, the HCD module reinitializes the host controller hardware.

When the host controller hardware has been reinitialized, the USB driver module unloads the USB device drivers loaded prior to the POWER_DOWN notification, identifies all the USB devices currently connected to the bus—a process called bus enumeration—and loads the USB device drivers for those devices. This suspend and resume processing is very similar to that performed by the Windows CE Device Manager for PC Card–based devices.

Bus-powered and Self-powered USB Devices

Windows CE 2.10 provides full support for bus-powered and self-powered USB devices. When a user connects a self-powered or bus-powered device to a Windows CE–based platform, the USB system software automatically accepts or rejects the device, based on the power requirements of the device. The power model is the same for both bus-powered and self-powered devices.

When a USB device is attached to a Windows CE–based platform, the HCD module sets the initial power configuration. During the device attachment processing phase, the HCD module reads the power requirements of the USB device configurations from the device configuration descriptor structures. In this way, the HCD module can choose an appropriate power configuration for the device.

Some devices may provide several configurations with different power requirements. OEMs who port an HCD module to their hardware can implement policies to choose the appropriate power configurations from those provided by the USB devices.

For example, Windows CE–based platforms have a registry setting that specifies the maximum total current draw allowable for USB devices connected to the host computer. If enabling a device would exceed this power threshold, the device is not configured unless the device has an alternate configuration with acceptable power requirements. OEMs can customize the platform-specific portions of the HCD module to choose dynamically whether to configure devices based on the current system power level. OEMs can implement a power model suitable for their platforms because the HCD module calls platform-specific code in its PDD layer for all USB devices connected to the bus. Therefore, an OEM can implement a power model that can selectively grant or deny power to individual USB devices according to whatever criteria the OEM chooses.

Because an HCD module cannot know which configuration might be appropriate for different uses of a USB device, a USB device driver can change its device configuration after the device driver is loaded, to the extent that the new configuration meets overall system power requirements. A USB device driver uses the **SetConfiguration** function to change a USB device configuration. In the unconfigured state, USB devices may not draw more than 100 mA.

USB Components Supplied by Microsoft

Microsoft supplies the following USB software components:

- The USB driver (USBD) module, which loads USB device drivers and manages resources in the USB subsystem.
- The complete set of USBD interface functions listed in the *Universal Serial Bus Specification, Revision 1.0* and implemented by the USBD module. The API includes transfer functions, pipe functions, device configuration functions, and miscellaneous functions. This API enables developers to write USB device drivers to support any USB-compliant devices.
- A sample HCD module that works with open host controller driver (OHCD)–compliant USB host controllers. OEMs must port this module to their hardware in order to support USB on their Windows CE–based platforms.
- A sample USB mouse device driver, which is loaded by the USBD module. The sample driver works with most generic USB mouse devices. Microsoft provides the sample for OEMs and IHVs to use solely as a reference while developing USB device drivers for their own USB devices.

Source code for these components is available in \Wince\Public\Common\Oak\Drivers\USB located in Windows CE Platform Builder. Header files are in \Wince\Public\Oak\Inc and \Wince\Public\Ddk\Inc. Platform sample code is in Platform\Cepc\Drivers\Usb.

Writing an HCD Module

To support USB on a Windows CE–based platform, OEMs must customize the HCD module included with the Platform Builder. Microsoft provides a sample HCD module that works with OHCD-compliant USB host controllers. The sample OHCD is designed for a Windows CE–based platform. OEMs must port the sample OHCD to their platforms as part of their OAL implementations.

To ease the porting effort, the OHCD is divided into two parts: a platform-independent MDD and a small PDD. This is similar to the MDD/PDD device driver model used by Windows CE–based native device drivers. As with the native device driver model, OEMs need to port only the PDD part. Furthermore, because the specifications for OHC interface and UHC interface hardware explicitly define the behavior of such hardware, the PDD layer does very little. The PDD layer for most HCD modules merely has to locate the hardware address of the host controller within memory and provide the MDD layer with the hardware address and a pointer to a shared memory area.

However, the MDD and PDD layers in the HCD module interact more than the MDD and PDD layers in most native device drivers. For example, in the majority of native device drivers, the MDD layer calls DDSI functions exposed by the PDD layer. In the HCD module, the MDD layer not only calls the DDSI functions, but also exposes a set of functions that the PDD layer must call at specific times during initialization.

OEMs must create a registry key within **HKEY_LOCAL_MACHINE\Drivers\Builtin\** so that the Device Manager loads the HCD module when the platform starts. Device drivers that the Device Manager loads must expose some of the stream interface functions. Therefore, the PDD layer is required to include some additional DDSI functions, such as **OhcdPdd_Open** and **OhcdPdd_IoControl**, even though the HCD module is not a stream interface driver. The benefit of this approach is that the Device Manager calls the HCD module's **OhcdPdd_Init** function, which in turn enables the HCD module to call the required MDD functions. The following example shows the registry key for the HCD module; it should contain the same values as other stream interface driver keys.

```
[HKEY_LOCAL_MACHINE\Drivers\Builtin\OHCI]
  "Prefix"="HCD"
  "Dll"="ohci.dll"
  "Index"=dword:1
  "Order"=dword:1
```

The following list shows the MDD functions that the PDD layer must call:

OhcdMdd_CreateMemoryObject

OhcdMdd_DestroyMemoryObject

OhcdMdd_CreateOhcdObject

OhcdMdd_DestroyOhcdObject

OhcdMdd_PowerUp

OhcdMdd_PowerDown

The following list shows the DDSI functions exposed by the PDD layer:

OhcdPdd_CheckConfigPower	**OhcdPdd_Open**
OhcdPdd_Close	**OhcdPdd_PowerDown**
OhcdPdd_Deinit	**OhcdPdd_PowerUp**
OhcdPdd_DllMain	**OhcdPdd_Read**
OhcdPdd_Init	**OhcdPdd_Seek**
OhcdPdd_IoControl	**OhcdPdd_Write**

OEMs must write their own HCD modules if their platforms contain universal host controller driver (UHCD)–compliant host controllers, instead of OHCD-compliant host controllers. An HCD module for UHCD hardware must expose the same interface to the USB driver module as the sample HCD module. The OHCD sample is suitable to use as a model while developing an HCD module for UHCD hardware.

Adding Drivers for Additional Built-in Devices

You can add hardware devices to your Windows CE–based target platform that are not directly supported by Windows CE. However, if you do, you must supply device drivers for the additional devices.

Windows CE currently supports several classes of devices through the GWES module. OEMs cannot add new device classes through this module because this would require adding code to the module to enable it to handle I/O events for the new device classes. The Platform Builder does not include source code for the GWES module, which means that only Microsoft can add support for new device classes through the module.

Because the GWES module is not extensible, OEMs must provide drivers for any new types of device by using the other Windows CE–based driver model, the stream interface driver model. Stream interface drivers are user-mode DLLs managed by the Device Manager. Unlike native device drivers, each of which has its own interface to the GWES module, all stream interface drivers expose the same basic interface to the kernel. This interface consists of a set of functions that presents the capabilities of a device in terms of standard file I/O functions.

Windows CE versions 1.01 and later provide kernel support to enable stream interface drivers to access additional built-in hardware devices. Stream interface drivers can call the **KernelIOControl** function, which provides a generic method for OEMs to expose new functionality on their Windows CE–based platforms. **KernelIOControl** passes its parameters to the **OEMIoControl** function in the OAL, which OEMs can use to provide whatever hardware access is necessary. The Platform Builder has complete documentation on **KernelIOControl** and on the OAL.

For more information, see "Writing a Stream Interface Driver DLL."

CHAPTER 3

Developing Stream Interface Device Drivers

Stream interface drivers are dynamic-link libraries (DLLs) that typically are loaded, controlled, and unloaded by a special application called the Device Manager. In contrast to native device drivers, which have special single-purpose interfaces, stream interface drivers all use the same interface and expose a common set of functions—the stream interface functions.

Stream interface drivers are designed for peripheral devices that are connected to a Windows CE–based platform. Examples of these devices are modems, printers, digital cameras, and PC Cards. All peripheral devices must be attached through external connectors, such as serial ports or PC Card sockets. Therefore, device drivers for peripheral devices are like printer drivers for desktop computers: both run as user-mode processes that access the services of built-in hardware to control their devices. Device drivers for serial devices use the serial port. Device drivers for PC Cards use the PC Card Services library; in turn, the PC Card Services library controls the PC Card socket hardware. Because of these interdependencies, a stream interface driver often uses a native device driver—specifically, the native device driver for the external connector to which the peripheral is attached—in order to interact with the peripheral. However, stream interface drivers can interact directly with a peripheral if that peripheral is mapped to some portion of the system's memory space, as do stream interface drivers for devices that are often built into a Windows CE–based platform, such as speakers and microphones.

The primary task of a stream interface driver is to expose the services of a peripheral to applications by presenting the device as a special file in the file system. This is similar to other operating systems, such as UNIX, which exposes peripheral devices by means of special files in the file system \Dev directory. Although the Windows CE operating system (OS) does not have a direct analog to the UNIX \Dev directory, the device files in Windows CE are stored in the \Windows directory, and a special naming convention differentiates the device files from other files.

Despite the generic characteristics of stream interface drivers, they can be implemented in different ways. For example, even though stream interface drivers typically are designed by independent hardware vendors (IHVs) for peripheral devices, OEMs who customize Windows CE–based platforms sometimes write stream interface drivers for certain built-in devices. In addition, although stream interface drivers are usually loaded and unloaded by the Device Manager, sometimes applications perform the loading and unloading.

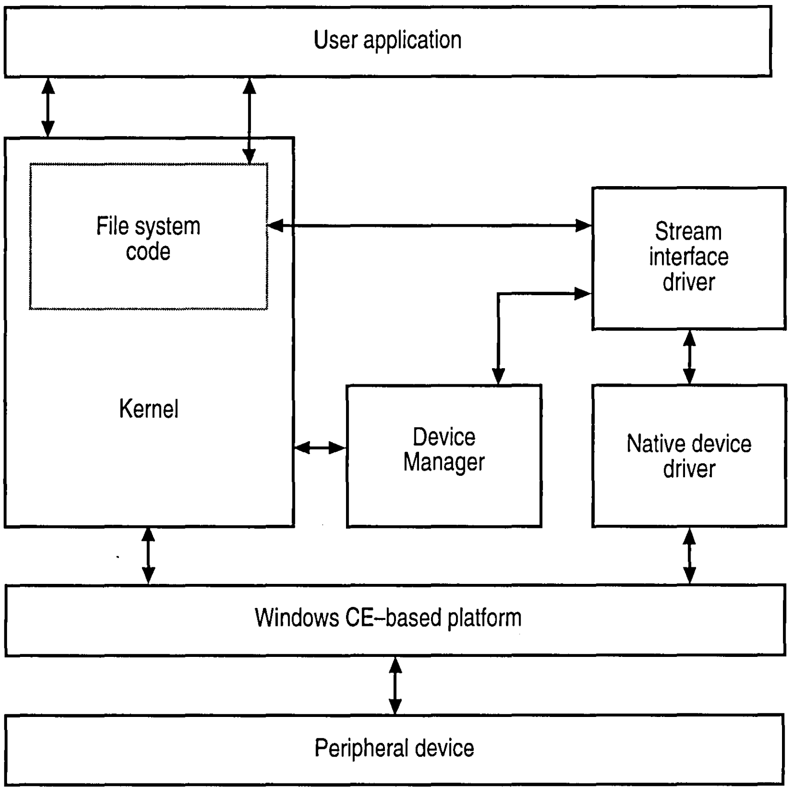
Application-specific stream interface drivers present resources in ways that an application can readily use. Peripheral devices are the most common resource that is used by application-specific stream interface drivers, but almost any resource, such as memory, process lists, or TCP/IP ports, can be presented in terms of a stream interface driver. For example, a company that makes Global Positioning System (GPS) receivers could make two versions of the receiver: one that is built into a PC Card and one that connects to an external serial port. Rather than adding complexity to the application that uses the GPS receiver so that the application can use either type of receiver, the company could write a device driver that repackages the services of one type of receiver in the interface of the other type. Thus, the application would not need to distinguish between the two types of receiver that are connected to the Windows CE–based platform; both types would react identically to the application despite the different access methods used for serial and PC Card devices.

The following sections describe how to write stream interface drivers in greater detail. Included are explanations of the system architecture for stream interface drivers, device file names, the Device Manager, entry points, and the relationship of a stream interface driver to a device and to applications.

System Architecture for Stream Interface Drivers

A stream interface driver receives commands from the Device Manager and from applications by means of file system calls. The driver encapsulates all of the information that is necessary to translate those commands into appropriate actions on the devices that it controls.

The following illustration shows the relationship between a stream interface driver and other system components.



The following definitions apply throughout the *Microsoft Windows CE Device Driver Kit*:

User application
: A user application includes any application that accesses a peripheral device. Access to a peripheral is through the file system and the special files that expose the services of peripherals.

Kernel
: The kernel, which is the core of the Windows CE OS, provides basic services to the application. To support stream interface drivers, the kernel redirects the file I/O function calls of the application to the appropriate entry points in the stream interface driver.

Device Manager
: The Device Manager is a special application that loads and unloads stream interface drivers. For more information, see "Device Manager."

Stream interface driver
: A stream interface driver is a DLL that manages a peripheral device. It exposes file I/O functions to applications and implements those functions by accessing its peripheral device to map the device's capabilities to the semantics of the file I/O functions. Stream interface drivers can either use an underlying native device driver to access the physical peripheral devices that the driver serves or access its device directly if the device is mapped into memory. Audio device drivers for built-in audio hardware are an example of direct access.

Native device driver
: A native device driver is a device driver that is supplied by the manufacturer of a Windows CE–based platform. Some stream interface drivers use native device drivers to access peripherals because peripherals are attached to Windows CE–based platforms through connectors that are managed by native device drivers. The native device drivers that are used by stream interface drivers are generally drivers for various sorts of I/O ports, such as serial ports or PC Card sockets.

Windows CE–based platform
: A Windows CE–based platform consists of a CPU, memory, clock, and possibly other hardware, such as a keyboard and touch screen. Stream interface drivers generally use the I/O ports on a platform, whereas the kernel can use all of a platform's hardware.

Peripheral device
: A device—such as a modem, printer, microphone, keyboard, touch screen, and so on—that is managed by a device driver.

Device Manager

This section discusses the Device Manager, its position within the Windows CE OS, and its management of stream interface drivers. Developers do not need in-depth knowledge of the Device Manager in order to write stream interface drivers, but a general understanding of the Device Manager and its role within Windows CE is useful.

The Device Manager, which is a user-level process, typically runs continuously on a Windows CE–based platform. The Device Manager is not part of the kernel, but it is a separate application that interacts with the kernel, the registry, and stream interface driver DLLs. More specifically, the Device Manager performs the following tasks:

- Initiates the loading of a driver at system startup or when it receives notification that a user has attached a peripheral to the Windows CE–based platform. For example, when a user inserts a PC Card, the Device Manager attempts to locate and load a device driver for that PC Card.

- Registers special file names with the kernel that map the stream I/O functions that are used by applications to the implementations of those functions within a stream interface driver.
- Finds the appropriate device driver for a peripheral by obtaining a Plug and Play identifier from the peripheral device or by invoking a detection routine to find a driver that can handle the device.
- Loads and tracks drivers by reading and writing registry values.
- Unloads drivers when their devices are no longer needed. For example, the Device Manager unloads a PC Card device driver when a user removes the card.

Registry Keys Used by the Device Manager

The Device Manager uses registry keys that are stored in the **HKEY_LOCAL_MACHINE\Drivers\** key. The following are the major subkeys of the **HKEY_LOCAL_MACHINE\Drivers\** key:

Active

The **Active** key contains subkeys that track currently active drivers that were loaded by the Device Manager. Device driver setup routines should not modify the **Active** key's contents, nor should they rely on the presence of any specific values within the **Active** key.

When a device driver is loaded, the Device Manager passes to the device driver the path to its **Active** key as the *dwContext* parameter in the device driver's *XXX*_**Init** function. The device driver can rely on the presence of the following values:

- **Hnd**

  Device handle from the **RegisterDevice** function
- **Name**

  Device file name
- **Key**

  Registry path within **HKEY_LOCAL_MACHINE** to the device driver key

The **Active** key for PC Card device drivers has the following additional values:

- **PnpId**

  Plug and Play identifier string
- **Sckt**

  Current socket and function pair of the PC Card

Once a device driver is loaded, it can add values to its **Active** key. However, the Device Manager deletes the device driver **Active** key and any values in it when the device driver is unloaded.

Builtin
The **Builtin** key contains subkeys that govern stream interface driver DLLs provided by the manufacturer of a Windows CE–based platform. As was stated previously, OEMs can implement drivers for certain types of built-in devices, by using the stream interface driver model.

PCMCIA
The **PCMCIA** key contains subkeys that are related to PC Cards and their stream interface drivers. The most important information in these subkeys is the Plug and Play identifier of the PC Card, which corresponds to a specified driver.

Detect
The **Detect** subkey contains numbered entries that list DLL names and detection functions. The functions identify a generic stream interface driver for PC Cards with no Plug and Play identifier, or for PC Cards with an unknown Plug and Play identifier.

Driver
Driver subkeys, named after generic PC Card drivers, contain values that are used to load the drivers. The Device Manager creates individual ***Driver*** subkeys when a generic stream interface driver detection function indicates that it can drive a particular device. The existence of a ***Driver*** subkey indicates detection by a driver that has an entry within the **PCMCIA\Detect\** key.

Plug-and-Play ID
Plug-and-Play ID subkeys contain values that are used to load stream interface drivers for PC Cards. Typically, the setup routine for a PC Card driver creates these subkeys when the driver is installed on a Windows CE–based platform.

Loading Stream Interface Drivers

There are three ways to load stream interface drivers. Two of them involve the Device Manager, but the third is specific to applications.

The first type of loading occurs at startup. When a Windows CE–based platform starts up, it starts the Device Manager. In turn, the Device Manager reads the contents of the **HKEY_LOCAL_MACHINE\Drivers\Builtin** key and loads any stream interface drivers listed there. For example, on many Windows CE–based platforms, the Device Manager loads the driver for built-in serial ports (Serial.dll) through this mechanism.

The second type of loading occurs when the Device Manager automatically detects the connection of a peripheral device to a Windows CE–based platform. A PC Card is the most common type of automatically detectable device because the

PC Card socket controller notifies Windows CE when a user inserts a PC Card. When a user inserts a PC Card into a socket, the Device Manager calls the socket driver, which is a native device driver, to find the Plug and Play identifier. The Device Manager then checks the **HKEY_LOCAL_MACHINE\Drivers\PCMCIA\** key for a subkey matching the Plug and Play identifier. If one exists, it loads the driver listed within that key. If there is no match, the Device Manager calls all of the detection functions that are listed within the **HKEY_LOCAL_MACHINE\Drivers\PCMCIA\Detect** key. If one of the detection functions returns a value indicating that it can handle the PC Card, the Device Manager loads and initializes that stream interface driver.

In the first and second types of loading, the Device Manager calls the **RegisterDevice** function for the stream interface driver and creates a numbered subkey within the **HKEY_LOCAL_MACHINE\Drivers\Active\** key to track the driver. **RegisterDevice** also locks the stream interface driver into working RAM. This prevents the driver from being swapped out and prevents any paging activity that would slow driver operation when servicing interrupts.

The third type of loading occurs when the Device Manager cannot automatically detect the connection of a peripheral to a platform. Unrecognized devices are often serial devices because there is no automatic way for Windows CE to detect the connection of a serial device to a serial port. If the Device Manager cannot automatically recognize a peripheral, the application that needs to use the peripheral must load the peripheral's driver. The following is the standard sequence of actions for this type of loading:

1. A user starts the application.
2. The application determines that the device driver is not loaded, either by inspecting the file system or by receiving a failure return value from the **CreateFile** function when it tries to open the device file name of the peripheral. For more information, "see Device File Names."
3. The application calls **RegisterDevice** to load the stream interface driver and lock it into memory.
4. The application continues with its usual operation.

Unloading Stream Interface Drivers

A stream interface driver DLL can be unloaded in two ways, depending on the method by which the DLL was loaded. If the Device Manager loaded the DLL, the Device Manager detects the disconnection of the stream interface driver's peripheral from the Windows CE–based platform, such as when a PC Card is ejected from its socket. At this point, the Device Manager removes the driver's entry from the **HKEY_LOCAL_MACHINE\Drivers\Active** key, calls the

DeregisterDevice function to remove the peripheral's device file name from the file system, and notifies the **FreeLibrary** function to unload the DLL. However, if an application loaded the DLL, the application must unload the DLL before it exits by calling **DeregisterDevice**.

Device File Names

Applications access peripheral devices through special entries in the file system. The file system code in the Windows CE OS includes code that recognizes these special file names and redirects file I/O operations to the appropriate stream interface driver.

This section provides information about the file name formats, prefixes, and indexes that you need to establish to develop stream interface drivers. For more information on how to use device file names when developing a stream interface driver, see "Writing a Stream Interface Driver DLL."

Device File Name Format

The file system recognizes file names as special device files if the file names consist of exactly three uppercase letters, a single digit, and a colon (:). This format follows the convention established in the Microsoft® MS-DOS® operating system for serial and parallel ports. For example, COM1:, PGR7:, and GPS0: are valid device file names, but MODEM1:, COM27:, and LPT1 are not.

Device File Name Prefixes

The prefix consists of three uppercase letters that identify which special device file name corresponds to a particular stream interface driver. The prefix is stored in a registry value called **Prefix**, which is located within the key for the driver. Typically, the setup utility that installs a driver creates this registry value, along with the other values that stream interface drivers need.

When you create a stream interface driver, you designate the three-letter prefix. It can be any three letters, although you should use a common prefix if your driver is for the same class of device as other drivers already present on the Windows CE–based platform. For example, drivers for serial devices, such as modems, could use the common prefix COM, even though other drivers might already be using that prefix. Your driver can distinguish itself from any others with the same prefix by using a different index.

Windows CE uses the prefix in two ways. First, the prefix identifies all possible device file names that can access the stream interface driver. Second, the prefix tells the OS what entry-point names to expect in the stream interface driver DLL. For example, to implement a device driver for a PC Card pager, you could choose PGR as the three-letter prefix, which in turn would dictate entry-point names, such as **PGR_Open**, **PGR_IOControl**, and so on.

Device File Name Indexes

In a special device file name, the index is the digit that follows the prefix. The index differentiates devices that are managed by a stream interface driver. By default, the Device Manager numbers indexes logically from 1 through 9, with 1 corresponding to the first device file name. If you require a tenth device file name, use 0 as the index.

If you need to number your device file names starting at an index other than 1, specify a starting index in a registry value called **Index** within the registry key for your driver. This is often necessary if your stream interface driver serves a device that should use a common prefix, such as COM. For example, on many Windows CE–based platforms, COM1:, COM2:, and COM3: correspond to built-in serial port hardware. If your driver is for a serial device, such as a packet-radio modem, it should appear as a COM port because modem software often assumes that modems are connected to COM ports. You could specify an **Index** value of 4 to differentiate your serial device from the ones that arebuilt into the hardware.

If you specify an index, rather than letting the Device Manager assign indexes as needed, by default your driver supports only one device because the Device Manager can register only one device file name. If you need to specify an index but need more than one device file name, you have two options: the *XXX*_**Init** function can register additional device file names with the **RegisterDevice** function, or the setup utility can create additional sets of registry keys, each with a different index, when your driver is installed.

Writing a Stream Interface Driver DLL

The following sections explain the important issues that affect your design decisions as you implement a stream interface driver DLL. Topics covered include how to create a DLL containing required entry points for stream interface drivers, how to regulate either single access or multiple access to a driver, and, for developers who implement interrupt-driven devices, how to plan for interrupt processing and real-time processing. In addition, this section offers specific information regarding drivers for serial devices and PC Card devices. The final section discusses the installation of a stream interface driver on a Windows CE–based platform.

Required Entry Points for Stream Interface Driver DLLs

For every stream interface driver DLL, the required entry points implement standard file I/O functions plus power management functions that are used by the Windows CE kernel. The following list shows the required entry points:

*XXX*_**Close**	*XXX*_**Deinit**
*XXX*_**Init**	*XXX*_**IOControl**
*XXX*_**Open**	*XXX*_**PowerDown**
*XXX*_**PowerUp**	*XXX*_**Read**
*XXX*_**Seek**	*XXX*_**Write**

When you create a DLL, replace *XXX* in the entry-point names with a device file name prefix.

Single Access and Multiple Access

Because peripheral devices are exposed to applications as special files when you create a stream interface driver, you provide the implementation of such a file. Therefore, consider whether it is practical, based on the capabilities of the device that your driver serves, to enable multiple applications to have simultaneous access to the special file; that is, consider whether your driver can have multiple open file handles on its device. A stream interface driver can implement either single access or multiple access by using the *hOpenContext* parameter that Windows CE passes to all file I/O functions.

To enable multiple access, each call to the *XXX*_**Open** function should return a different value for *hOpenContext*. The device driver must track which return values from *XXX*_**Open** are in use. Subsequent calls by Windows CE to *XXX*_**Close**, *XXX*_**Read**, *XXX*_**Write**, *XXX*_**Seek**, and *XXX*_**IOControl** pass these values back to the device driver, enabling the driver to identify which internal data structures to manipulate.

To enforce single access, only the first call to *XXX*_**Open** should return a valid *hOpenContext* value. As long as this value remains valid, which is until Windows CE calls *XXX*_**Close** for the value, subsequent calls to *XXX*_**Open** should return NULL to the calling application to indicate failure.

Interrupt Processing

Some peripheral devices can cause or signal interrupts on the bus of a Windows CE–based platform. Typically, the peripherals are PC Cards. Some Windows CE–based platforms use standard buses, such as the Peripheral Component Interconnect (PCI) local bus, in which case expansion cards plugged into the bus can also signal interrupts. Because these peripheral devices can cause or signal interrupts, their stream interface drivers need code to process interrupts.

The recommended method for processing interrupts is for the stream interface driver to spawn a new thread, called an interrupt service thread (IST), during its processing of the *XXX*_**Init** call. When an interrupt occurs, Windows CE signals the IST. The IST interacts with the peripheral device to perform whatever device-specific processing is necessary to handle the interrupt. This method is beneficial because Windows CE versions 2.12 and earlier do not support nested interrupts. Thus, while an interrupt service routine (ISR) is running, other interrupts are masked for a short time. ISRs should therefore do as little processing as possible, and should report only the logical interrupt value to the OS so that Windows CE can activate the corresponding IST. By separating the interrupt processing into a very short ISR and a longer user-mode IST, interrupts can be masked for as little time as possible.

ISTs generally run at a high priority so that they can respond quickly to interrupt events. For most drivers, this thread should register to receive interrupt notifications from the **InterruptInitialize** function. For PC Card drivers, however, the thread receives notifications from the **CardRequestIRQ** function. PC Card drivers do not process interrupts directly because the built-in PC Card socket driver gets the raw interrupts that are generated by the PC Card socket. **CardRequestIRQ** allows you to specify a callback function that the PC Card socket driver calls when an interrupt occurs.

Real-Time Processing

Hard real-time processing performance of Windows CE versions 2.12 and earlier varies widely according to processor and bus speed. Windows CE versions 2.12 and earlier do not support nested interrupts, which greatly limits real-time processing. On the Windows CE–based platform, which uses the Hitachi SH3 microprocessor, interrupt-driven device drivers are typically started within 90 to 170 microseconds after the interrupt occurs, yielding theoretical sampling rates of about 5.8 to 11 kilohertz (kHz). The variability is due to factors such as what data is currently in the processor cache and whether the device driver process happens to be the one currently executing. On other hardware, factors such as CPU speed, bus speed, and the speed of the manufacturer's interrupt vectoring routines determine the lower limits of interrupt latency. Because the ISTs of device drivers can be preempted by high-priority threads, there is no absolute upper limit. In general, however, the latency for servicing interrupts in Windows CE is less than the latency for Windows-based desktop platforms; device drivers are unlikely to lose data unless they are starved for processor time by other high-priority threads running on the OS. Device drivers for polled devices may be able to achieve higher sampling rates, because polled devices do not generate interrupts and are therefore not subject to the same latencies.

Peripherals that connect to a Windows CE–based platform through a serial port can do so at the maximum speed of the serial port. Most serial ports are buffered internally by 16550-class universal asynchronous receiver-transmitters (UARTs) capable of relaying 115 kilobits per second; consult the manufacturer for specific information. PC Card socket speeds vary also; again, the authoritative source of information is the manufacturer.

To reduce the time that it takes your stream interface driver to handle incoming data, defer complicated processing. If the device driver restricts itself to collecting the data in memory, it can leave processing of that data for later, to be handled by a low-priority thread or user-level application.

Drivers for Serial Devices

Windows CE supports two methods of driving serial devices. The first requires you to create a stream interface driver DLL that presents high-level information from the device to applications. The second does not require that you write a driver; instead, it requires the applications that use the device to interpret the device data.

These two methods are possible because serial devices are always accessed through built-in COM ports, generally COM1: through COM3:. With the first method, you implement a serial device driver to present a new device file name to applications, such as COM4: or another file name specific to the function of the serial device. Internally, a device driver uses the services of the built-in COM port to access the peripheral. You can also let user applications open a built-in COM port to access the peripheral device directly.

The following are the factors allowing you to decide between these two strategies:

- Single access or multiple access

  If the peripheral can support simultaneous access by multiple applications, implement a stream interface driver. This greatly simplifies the user-level applications.

- Complexity of the incoming data stream

  If the incoming data stream from the peripheral is complicated and requires considerable processing to yield usable information, you probably should write a stream interface driver.

- Speed of the incoming data stream

  If data comes from the peripheral at a very high rate, there might not be enough processing time available to support the overhead that is required for a stream interface driver. In this case, you might have no choice but to put all processing into the user-level application.

Drivers for PC Card Devices

Stream interface drivers for PC Cards are more complex than drivers for serial devices because PC Cards themselves tend to be more complex. Stream interface drivers access peripheral devices through built-in hardware. For PC Cards, the built-in hardware is the PC Card socket or sockets available on a Windows CE–based platform. Specifically, these sockets are PC Card Type II sockets. The sockets are driven by the built-in PC Card socket driver, which implements the socket functions that you use to write stream interface drivers for PC Cards.

Most PC Card socket functions have a parameter that represents a socket and function pair. A socket and function pair is a combination of one particular PC Card socket and one particular function of a PC Card. Socket and function pairs support multifunction PC Cards and platforms with more than one PC Card socket. Drivers for multifunction cards should register one device file name for each function. Any drivers that you create for PC Cards should also be written to work with any PC Card socket because different platforms have different numbers of sockets.

If you write a generic PC Card driver—one that can drive an entire class of PC Cards adhering to a specified operating standard—you need an additional entry point in the DLL for a detection function. The Device Manager uses the detection function when a user inserts an unknown PC Card with no Plug and Play identifier. This function must conform to the PFN_DETECT_ENTRY prototype declared in Public\Common\DDK\Inc\DevLoad.h. For more information on PC Card detection functions, see "Device Manager."

Installing a Stream Interface Driver DLL

Installing a stream interface driver DLL on a Windows CE–based platform is a relatively simple procedure. However, you should provide a setup application for users because proper installation of most stream interface drivers requires changes to the registry. The setup application runs on a host computer and connects to a Windows CE–based platform through a serial cable or similar connection.

The setup application must perform the following steps:

1. Connect to the Windows CE–based platform or detect that a connection already exists.
2. Copy the stream interface driver DLL into the \Windows directory of the platform.
3. Create registry keys and values for the driver, if necessary.

For information about connecting a host computer to a Windows CE–based platform and copying files from a host computer to a Windows CE–based platform, see the Windows CE Platform SDK. For information about required registry keys for a stream interface driver, consult the *Microsoft Windows CE API Reference* included in this documentation.

Sample Stream Interface Drivers

The Platform Builder contains sample device drivers that illustrate how to write stream interface drivers for several types of device. If you implement a stream interface driver for a device similar to one of the samples, you can base your driver on the appropriate source code of the sample driver. The following table shows the sample device drivers contained in the Platform Builder.

Sample	Description
Pager Card	Illustrates a device driver for the Motorola NewsCard pager
Modem Card	Illustrates a device driver for a PC Card modem
RAM Card	Illustrates a device driver for a Static RAM PC Card
PC Card Test	Illustrates a device driver for testing PC Cards
Serial	Illustrates a device driver for devices that connect to a serial port
Touch Screen	Illustrates a device driver that interacts with the screen

The following are PC Card drivers:

- 16550mod

  A sample client driver for an I/O-type PC Card. A sample application included with this driver opens, writes, reads, and closes a device by making calls to the 16550MOD driver.
- Pager

  A sample pager PC Card driver supporting the Motorola NewsCard pager. The Platform Builder includes a sample application that enables users to receive and view pages.
- Ramcard

  A sample driver for a memory-type PC Card. The Platform Builder includes two sample applications for writing to and reading from the card.
- Cardtest

  A sample driver and test for a block of memory on a PC Card memory card or to dump attribute memory or tuple information from a PC Card.

The following are serial port drivers:

- GPS

  A sample serial-based client PC Card driver based on a GPS PC Card.
- TTY

  A sample serial-based client PC Card driver based on a sample teletypewriter (TTY) terminal.

For network driver interface specification (NDIS) drivers, NDIS sources consist of source code that is compatible with Windows 95 and Windows NT Workstation. The source code can simply be recompiled as a Windows CE–based DLL. For more information, see "NDIS Network Drivers."

CHAPTER 4

Audio Drivers

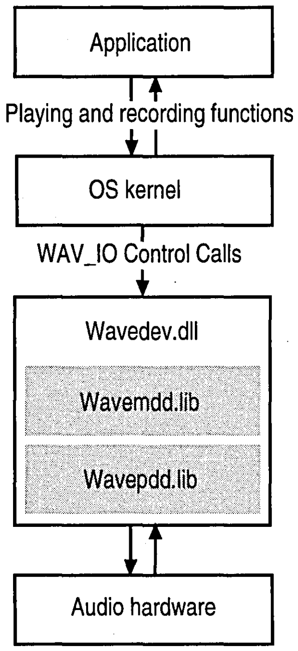
A waveform audio driver is responsible for processing messages from the Wave API Manager to play and record waveform audio. Waveform audio drivers are implemented as stream interface drivers that are loaded by the Device Manager. The sample waveform audio driver is named WaveDev.dll. All audio drivers use the prefix WAV to name their stream interface functions, yielding function names such as **WAV_Open**, **WAV_IOControl**, and so on.

Audio hardware typically supports a larger set of operations than usually apply to files. For example, files commonly do not have volume controls or playing speed controls, but audio hardware can and often does have them. The **DeviceIOControl** portion of the stream interface allows arbitrary operations on files, making it possible to manipulate audio hardware by means of **WAV_IOControl**. To send commands to the audio hardware, the operating system passes various messages to this function. For example, to prepare the audio hardware for recording, Waveapi.dll uses **WAV_IOControl** to send the WIDM_PREPARE message to the audio driver. The messages sent to the audio driver are similar to those used by user-mode audio drivers on Windows-based desktop platforms, such as Mmdrv.dll.

Because audio drivers rely entirely on **DeviceIOControl** function messages, the implementation of the remainder of the stream interface is relatively simple. Specifically, the **WAV_Read**, **WAV_Seek**, and **WAV_Write** functions are merely stubs that return constant values. The other stream interface functions should be fully implemented and follow the conventions described in the *Microsoft Windows CE API Reference* for those functions.

As an alternative to implementing the stream interface directly, you can use the model device driver (MDD) library—Wavemdd.lib—supplied by Microsoft. This library implements the stream interface functions in terms of the audio device driver service-provider interface (DDSI) functions. If you use Wavemdd.lib, you must create a matching platform-dependent driver (PDD) library that implements the audio DDSI functions. The PDD library is generally called Wavepdd.lib, although there is no requirement that it be called this. These two libraries can then be linked together to form Wavedev.dll.

The following illustration shows the interaction of the audio driver with the Windows CE operating system (OS).



As shown in the illustration, the first step in playing and recording sounds is a call from an application to the OS. The OS translates such calls into **WAV_IOControl** calls to the audio driver. The OS component that performs this translation is the Wave API Manager. The audio driver then executes the appropriate actions on the hardware. The Device Manager only loads and registers the audio driver at startup time. It is not directly involved in the operation of the driver.

Like standard stream interface drivers, the audio driver uses registry keys both to store configuration information and to advertise itself to the OS. If the audio driver is not already listed in Platform.reg, create a key called **HKEY_LOCAL_MACHINE\Drivers\Builtin\Audio** to store configuration information. The Device Manager creates a key in **HKEY_LOCAL_MACHINE\Drivers\Active** for the audio driver when the driver is loaded at startup time. Because the Device Manager only checks for audio drivers at startup time, drivers for add-on audio hardware still need to have their registry entries in the **\Builtin\Audio** part of the registry, although their hardware is not built into any Windows CE–based platform.

The following list shows the stream interface functions for the audio driver:

WAV_Close	**WAV_PowerDown**
WAV_Deinit	**WAV_PowerUp**
WAV_Init	**WAV_Read**
WAV_IOControl	**WAV_Seek**
WAV_Open	**WAV_Write**

For more information, see "Developing Stream Interface Device Drivers."

The following list shows the DDSI functions for the audio driver:

PDD_AudioDeinitialize
PDD_AudioGetInterruptType
PDD_AudioInitialize
PDD_AudioMessage
PDD_AudioPowerHandler
PDD_WaveProc

Sample Audio Driver

The sample audio driver is a layered, interrupt-driven stream interface driver that drives a built-in audio device. The layers are the typical MDD and PDD layers that native device drivers use, although the device driver interface (DDI) functions in the MDD layer are the same as the stream interface functions that are used by ordinary stream interface drivers. This code is provided as a convenience and does not imply any restrictions in the way that audio hardware for Windows CE must function. An audio driver could be implemented that is monolithic, is polled, and that drives a peripheral device, if a user wants to do so. The sample audio driver MDD layer implements a single audio device capable of playing and/or recording pulse code modulation (PCM) waveform audio. The sample supports simultaneous recording and playing but may also be used with audio devices that can perform only one of those functions at a time. The PDD layer is responsible for communicating with the audio hardware to start and stop playing and recording and to initialize and deinitialize the hardware.

Playing and Recording

The basic operation for playing involves an interrupt and two direct memory access (DMA) buffers for output data. The Wave API Manager prepares the data blocks and sends them to the audio driver's MDD layer. The MDD checks to see if the block has data to be played. If so, it calls the **PDD_WaveProc** function by using the WPDM_START message. The PDD is responsible for copying data from the block to the temporary DMA buffers. Then, the DMA is started to play the data. When the first of the two buffers is finished playing, an interrupt occurs. The PDD should report the interrupt to be of type AUDIO_STATE_OUT_PLAYING. If there is more data to be played, the MDD sends the message WPDM_CONTINUE. Otherwise, it sends WPDM_ENDOFDATA. As data blocks finish playing, the MDD marks them as done and sends them back to the Wave API Manager.

The PDD should alternate between the two DMA buffers, filling one buffer as the other is playing. If the reserve buffer is not full before the first buffer's data is finished playing, the PDD layer should report this with the return value AUDIO_STATE_OUT_UNDERFLOW. This gives the MDD layer an opportunity to synchronize and continue playing.

Recording uses a mechanism that is similar to that used by playing, but in reverse order. Recording starts with the PDD message WPDM_START as before, but no data is copied until the first DMA buffer is full. Then, the PDD copies the data from the DMA buffer into the user block. Again, as blocks are filled, the MDD returns them to the Wave API Manager. Recording continues until the user application requests a stop and sends the WPDM_STOP message.

Interrupt Handling in Audio Drivers

The MDD layer is responsible for interrupt handling. It creates and manages the audio driver's interrupt service thread (IST). The IST calls the **PDD_AudioGetInterruptType** function to determine the cause of the audio interrupt. Based on the interrupt type, the MDD either performs a callback to the Wave API Manager or sends more data or buffer space to the PDD to be played or recorded.

The PDD layer can support simultaneous playing and recording with one interrupt by merging the recording and playing status in the AUDIO_STATE return value for **PDD_AudioGetInterruptType**. The value can represent a state change of either the playing or recording hardware, or both. The lower four bits of the return code represent the input status, and the upper four bits represent the output status. If the current interrupt is for both recording and playing, the values can be combined using the logical **OR** operator. However, if the interrupt requires only one operation, the other four bits should remain 0; otherwise, the MDD interprets this as a change or update in the hardware's status.

If your audio device uses more than one hardware interrupt—typically, one for playing and one for recording—the driver's interrupt service routines (ISRs) should both return the interrupt identifier specified by the PDD layer. When **PDD_AudioGetInterruptType** is called to determine the cause of the interrupt, the function can access the hardware to determine the actual playing or recording status for each interrupt event. If this mechanism is not suitable for your audio device, you must modify the sample MDD layer to handle multiple virtual interrupts and multiple ISTs.

Power Management in Audio Drivers

When the system goes into the suspend state, the Device Manager calls the audio driver's **WAV_PowerDown** function. This function should call the **PDD_AudioPowerHandler** function to allow the PDD to put the hardware in a low-power consumption mode, if it is available. When the system resumes, the Device Manager calls the audio driver's **WAV_PowerUp** function, which should also call **PDD_AudioPowerHandler. WAV_PowerUp** also calls the **SetInterruptEvent** function for the audio interrupt to allow any active handles to be released by the IST.

Audio Compression Manager Drivers

Audio Compression Manager (ACM) drivers are stream interface drivers. Like other stream interface drivers, ACM drivers are controlled by the Device Manager and they expose standard stream I/O functions. However, unlike other stream interface drivers, applications do not use these drivers directly. ACM drivers are called by the ACM, which sends messages to ACM drivers when applications need to play or record sounds.

There are several key reasons to write an ACM driver:

- To support a particular audio file format. For example, the .wav files on Windows-based platforms require ACM drivers that differ from ACM drivers for the .au files of Sun Microsystems. Similarly, audio files in other formats need customized ACM drivers.
- To convert audio file format to another format. The ACM driver application programming interface (API) can be used to load an audio file into a common format, which ACM drivers can then translate into their own format.
- To use an encoding or compression algorithm tailored to the characteristics of a particular type of audio. For example, human speech has a relatively limited frequency range and many essentially silent periods. An ACM driver written with these characteristics in mind can produce recordings that require little memory and yet retain good sound quality.
- To perform filtering of audio data. For example, an ACM driver could be used to filter a data stream out of an audio signal that contains speech in the lower frequencies and data in the higher frequencies.

An individual ACM driver can be written to perform a combination of these tasks. All ACM drivers must use ACM as their device file name prefix. In Windows CE, the **Index** keys in the registry settings designate individual ACM drivers. Thus, multiple ACM drivers have special device file names such as ACM1:, ACM2:, and so on. A maximum of eight ACM drivers can coexist on a Windows CE–based platform; **Index** value 9 is reserved for the PCM converter exported from Waveapi.dll.

The ACM uses **Index** keys when searching for an ACM driver to perform format conversions. The ACM selects the first appropriate ACM driver that it finds. For more details about device file names, and **Index** keys, see "Developing Stream Interface Drivers."

ACM drivers respond to messages similar to those used by ACM drivers written for Windows NT. In fact, the Windows CE–based ACM functions and structures that ACM drivers use are the same as those described in the Microsoft Windows NT Device Driver Kit. In Windows CE, the ACM uses the **ACM_IOControl** function to send messages to the driver. By calling this function, the ACM bypasses the three primary stream I/O functions: *XXX*_**Read**, *XXX*_**Write**, and *XXX*_**Seek**. Consequently, those functions are never called in an ACM driver. The ACM uses the other stream I/O functions only for setup and shutdown tasks.

The Microsoft Windows NT Device Driver Kit contains several sample ACM drivers. You can port these drivers to work on any Windows CE–based platforms that your product targets, and you can modify the drivers to support additional audio format types and audio filter types. The Microsoft Windows NT Device Driver Kit contains complete documentation on writing ACM drivers. For more information, see "Porting a Windows NT ACM Driver to Windows CE."

Types of ACM Driver

There are three types of ACM drivers:

- Codec

  Codecs convert one audio format type to another format type; typically, a compressed format to an uncompressed format. A codec can convert a compressed format, such as MS-ADPCM, to an uncompressed format, such as PCM, which most audio hardware can play directly.

- Converter

  Converters change one variety of a format to another variety of the same format. For example, a converter can convert a PCM audio stream sampled at 44 kilohertz (kHz) to a PCM audio stream sampled at 11 kHz.

- Filter

  Filters modify audio data without changing the format of the data. Filters generally are used to add some sort of audio effect to an audio stream. Tasks such as graphic equalization or adding an echo to an audio stream are appropriate for filters.

ACM Format Tags and Filter Tags

Format tags represent the names of individual formats, such as PCM. Filter tags represent the names of individual filters. Typical ACM drivers support one or more types of format or filter, or a combination of both. A set of formats or filters is associated with each of an ACM driver's tags. For example, the sample IMAADPCM driver, a codec, supports the WAVE_FORMAT_PCM format tag for the PCM format type and WAVE_FORMAT_IMA_ADPCM for the IMAADPCM format type. For each format tag, the driver supports a set of individual formats, consisting of combinations of sample rates and sample sizes.

The main difference between codecs and converters is that codecs transform data from a format belonging to one format tag into a format belonging to another format tag, whereas converters transform data between two formats that belong to the same format tag.

The supported format tags and filter tags are declared in the Mmreg.h header file. If you write an ACM driver for a new format type or filter type, register your driver with Microsoft.

ACM Driver Stream I/O Functions

The following table shows the ACM functions and their associated calls.

Function	Description
ACM_Init	Called by the Device Manager when the driver is loaded
ACM_Deinit	Called by the Device Manager when the driver is unloaded
ACM_Open	Called by the ACM when the driver is opened for use
ACM_Close	Called by the ACM when the driver is no longer needed
ACM_Read	Never called
ACM_Write	Never called
ACM_Seek	Never called
ACM_PowerUp	Called by the Device Manager when the system resumes from the suspend state
ACM_PowerDown	Called by the Device Manager when the system enters the suspend state
ACM_IOControl	Called by the ACM to pass messages to and receive information from the driver

ACM Driver Messages

The ACM passes the following messages to ACM drivers:

ACMDM_DRIVER_ABOUT
: Requests that the driver display its **About** dialog box.

ACMDM_DRIVER_DETAILS
: Requests that the driver return detailed information about its capabilities.

ACMDM_DRIVER_NOTIFY
: Notifies the driver of changes to other ACM drivers.

ACMDM_FILTER_DETAILS
: Requests information about a filter that is associated with a specified filter tag.

ACMDM_FILTERTAG_DETAILS
: Requests information about a specified filter tag.

ACMDM_FORMAT_DETAILS
: Requests information about the format that is associated with a specified filter tag.

ACMDM_FORMAT_SUGGEST
: Requests that the driver suggest a target conversion format for a specified source format.

ACMDM_FORMATTAG_DETAILS
: Requests information about a format tag.

ACMDM_HARDWARE_WAVE_CAPS_INPUT
: Requests information about the input capabilities of the audio hardware.

ACMDM_HARDWARE_WAVE_CAPS_OUTPUT
: Requests information about the output capabilities of the audio hardware.

ACMDM_STREAM_CLOSE
: Requests that the driver close a conversion stream.

ACMDM_STREAM_CONVERT
: Requests that the driver convert an audio stream.

ACMDM_STREAM_OPEN
: Requests that the driver open a new audio stream.

ACMDM_STREAM_PREPARE
: Requests that the driver prepare any buffers that are associated with an audio stream.

ACMDM_STREAM_RESET
: Requests that the driver stop operations on an audio stream.

ACMDM_STREAM_SIZE
: Requests that the driver return the size that is required for a source or destination buffer.

ACMDM_STREAM_UNPREPARE
Requests that the driver clear any prepared buffers that are associated with an audio stream.

Porting a Windows NT ACM Driver to Windows CE

To port an ACM driver from Windows NT to Windows CE, link the driver with the Acmdwrap.lib file, which provides the driver with the appropriate stream I/O interface. This library not only handles all interactions with the ACM and the Device Manager, but also passes messages from its **ACM_IOControl** function to the Windows NT ACM driver's existing **DriverProc** function.

CHAPTER 5

Printer Drivers

Windows CE versions 2.0 and later include support for printing. The printing model used by the Windows CE operating system (OS) is a subset of the printing model defined for desktop Windows-based platforms. Only a small number of the graphics driver functions defined for printer drivers are required in printer drivers for Windows CE.

The following sections describe the interaction between Windows CE–based printer drivers and Windows CE–based display mechanisms. These sections also describe the functions that printer drivers must implement and the port monitor functions that printer drivers call to send data to printers.

The Windows CE graphics display interface (GDI) and display driver perform most of the work involved in printing. At the beginning of the printing process, the GDI creates a device context with attributes that are retrieved from the printer driver during a call to the **DrvEnablePDEV** function. The display driver is used to render subsequent drawing commands that are issued from the application into the DC; the printer driver does not render the document. Then, the GDI sends the resulting bitmap to the printer driver so that the printer driver can format it and send it to the printer. To conserve memory, the GDI renders the document in bands, which are horizontal sections of the page, and the GDI makes several calls to the printer driver to send the rendered bands to the printer.

Windows CE–based printer drivers are required to implement only those graphics driver functions that are necessary for gathering printer metrics, setting up the printer, starting and ending print jobs, and preparing rendered strips for printing. Internally, the printer driver converts the bitmap data from a GDI bitmap format into the format that is required by the printer. This can include operations such as color reduction to the color space of the printer, data compression, and data conversion into the format that is used by the printer, a format sometimes known as a printer description language. Finally, the printer driver calls the port monitor to send the rendered strips to the printer.

Windows CE–based printer drivers are compiled as dynamic-link libraries (DLLs). As a result, they must export the **DrvEnableDriver** function.

For more information about Windows CE–based printer drivers, see "Functions Implemented by Printer Drivers" and "Registry Keys for Printer Drivers."

Functions Implemented by Printer Drivers

The following table shows the functions that Windows CE–based printer drivers must implement.

Function	Description
DrvCopyBits	Called by the GDI to copy a rendered band to the printer driver. During the call to the **DrvEnablePDEV** function, the printer driver specifies the bitmap format that is used in the call to this function.
DrvDisablePDEV	Frees memory and resources that are used by the driver when the printer device context is no longer needed.
DrvDisableSurface	Called by the GDI to inform the printer driver that the surface created for the current printing device context is no longer needed.
DrvEnableDriver	Receives two callback function pointers from the GDI and returns GDI function pointers for the other entry points that are implemented by the printer driver. This function is the entry point for the printer driver DLL, which must be exported in the .def file for the DLL.
DrvEnablePdev	Used by the GDI to gather device metrics for the target printer. The printer driver returns device specifics in the **GDIINFO** structure.
DrvEnableSurface	Creates a surface for use in rendering by calling the **EngCreateDeviceSurface** function.
DrvEndDoc	Called by the GDI to finish or abort a print job.
DrvGetModes	Returns information to the GDI about the default printing mode that is supported by the printer driver. The printer driver returns the default configuration in the **DEVMODE** structure.
DrvStartDoc	Called by the GDI to start a print job.
DrvStartPage	Called by the GDI to start printing the next page of a print job.

Port Monitor Functions Used by Printer Drivers

The port monitor manages all communication between printer drivers and physical communication ports on a Windows CE–based platform. Therefore, the printer driver does not need to support different types of printer ports explicitly; due to the port monitor's management, the driver automatically works with any type of printer port. The port monitor supports serial ports, infrared ports, parallel ports, and network printing. Printer drivers should call the port monitor application programming interfaces (APIs), instead of calling low-level communication APIs directly.

The following table shows the port monitor functions.

Function	Description
GetPrinterInfo	Retrieves information about the capabilities of a printer
PrinterClose	Closes a handle to a printer
PrinterOpen	Opens a handle to a printer
PrinterSend	Sends blocks of data to a printer

Registry Keys for Printer Drivers

Windows CE uses registry keys to store both global printer settings for the device and configuration information for individual printer drivers. The keys are stored under **KEY_LOCAL_MACHINE\Printers\** in the registry. For information on registry keys that relate to the default printer and to the ports available for printing, see "Global Printer Settings." For information on registry keys for specific printer drivers, see "Printer Driver Settings."

Global Printer Settings

The global printer settings that are stored in the registry list the I/O ports to use for printing, provide a global time-out value, and store which printer driver currently is selected as the default. The default printer driver is stored as a value for **DefaultPrinter**. The printer ports are stored as values in the **\Ports** subkey.

The following registry file excerpt defines five printer ports, two time-out values, and the default printer driver.

```
[HKEY_LOCAL_MACHINE\Printers\Ports]
  "Port1"="COM1: 9600"
  "Port2"="COM1: 57600"
  "Port3"="IRDA"
  "Port4"="LPT1:"
  "Port5"="NET0:"
[HKEY_LOCAL_MACHINE\Printers\Settings]
  "TimeOut"=dword:1E
[HKEY_LOCAL_MACHINE\PrintSettings]
  "TimeOut"=dword:2D
[HKEY_LOCAL_MACHINE\Printers]
  "DefaultPrinter"="PCL Laser"
```

The values correspond to serial ports at 9,600 baud and 57,600 baud, an infrared port, a parallel port, and a network printer port. For the two COM1: ports, a baud is also specified. The port monitor parses the values to extract the baud and device file name. In Windows CE 2.0, COM3: is assumed to be an infrared port. However, in Windows CE 2.10 and later the new file name "IRDA" is reserved for the infrared port, whereas COM3: is used for an additional serial port. When assigning an infrared port for printing, the value of the key for the port must be the literal string "IRDA," even though this value is not in the standard format for special device file names. The names "LPT1:" and "NET0:" are reserved for parallel and network printing, respectively, although the device file name indexes in those names can be any valid value.

Windows CE versions 2.02 and later use the **TimeOut** key. This value is a hexadecimal number that is measured in seconds; the example is for a 30-second time-out. After the time-out period, the printer driver can display a dialog box to give the user the option to retry or cancel. In this example, the default printer driver, "PCL Laser", is the name of a subkey within **HKEY_LOCAL_MACHINE\Printers\** where Windows CE can find settings for the Printer Control Language (PCL) laser printer driver.

For a printer that does not use the parallel port, such as an infrared or serial printer, it is difficult for Windows CE to determine whether the printer has received all the data to be printed. Therefore, a time-out parameter is used to detect and report printing errors if the printer fails to give any signal that it is still active. This time-out value is stored in the **PrintSettings\TimeOut** key. The default value is 45 seconds.

The value stored in the **DefaultPrinter** key lists the printer that is preselected in the common print dialog box that applications display to users.

Printer Driver Settings

Settings for individual printer drivers are stored in subkeys of the **HKEY_LOCAL_MACHINE\Printers\** key. The registry key for a printer driver should define four keys: **Driver**, **High Quality**, **Draft Quality**, and **Color**. The **Driver** value names the DLL that contains the printer driver. The **High Quality** and **Draft Quality** values store the resolution of high-quality and draft-quality modes, if the printer supports them. Not all printers support a draft-quality mode, so the **Draft Quality** key may be omitted. The **High Quality** key must always be present, and it corresponds to the highest-quality mode of the printer. The **Color** value defines whether a printer can print in color or only in monochrome. If color is supported, this key must be set to the literal string "COLOR" so that the common print dialog box enables color printing for applications.

The following example shows settings for the PCL laser printer driver and the PCL Inkjet printer driver.

```
[HKEY_LOCAL_MACHINE\Printers\PCL Laser]
  "Driver"="pcl.DLL"
  "High Quality"="300"
  "Draft Quality"="150"
  "Color"="Monochrome"
  "Version"="0x200"

[HKEY_LOCAL_MACHINE\Printers\PCL Inkjet]
    "Driver"="pcl.dll"
    "High Quality"="300"
    "Draft Quality"="150"
    "Color"="Monochrome"
```

In addition, the **Version** key can be used to support printer drivers that serve multiple printers.

CHAPTER 6

Display Drivers

The display device driver interface (DDI) is a subset of the Windows NT display DDI. If you are unfamiliar with the Windows NT display DDI, you may find it helpful to read the display driver sections of the Microsoft Windows NT Device Driver Kit before writing your Windows CE–based display driver.

The Windows CE operating system (OS) uses only the basic graphics engine functions and driver functions from the Windows NT display DDI. The differences between Windows CE and Windows NT have the following ramifications on Windows CE–based display drivers.

- Windows CE–based display drivers always present the same functionality; therefore, the graphics device interface (GDI) does not query a driver for information about its capabilities.
- A Windows CE–based display driver cannot reject an operation as too complex, and then call back into GDI to have the operation broken into simpler primitives. Because all Windows CE–based display drivers support the same functionality, the GDI can break up complex operations before calling the display driver.
- Windows CE–based display drivers are compiled as dynamic-link libraries (DLL) rather than libraries.

In Windows CE version 2.0 and later, the GDI makes calls to the display driver, and the display driver writes to the physical display device. All Windows CE–based display drivers must implement a set of display DDI functions, which are used to initialize the display driver and draw to the display.

Most Windows CE display drivers use a set of C++ classes called the Graphics Primitive Engine (GPE). The GPE classes serve as a base of code that you can use to derive new display drivers for your hardware. The GPE classes handle all communication within the display DDI layer. Using the GPE classes can save a large amount of development and debugging work. The S3Virge and all other sample display drivers included in the Microsoft Windows CE Platform Builder use the GPE. They demonstrate how you can add extensions to the display driver; for example, to enable hardware accelerations.

Display Driver Interface

In Windows CE, display drivers are native drivers because there is a custom interface between them and the Graphics, Windowing, and Events subsystem (GWES). They are loaded and called directly by their parent process, which can be either the GWES module or the Device Manager. Windows CE display drivers are most commonly written by using a layered architecture. The GPE library provided by Microsoft handles all of the default drawing, acting as the display driver's model device driver (MDD) layer. OEMs or independent hardware vendors (IHVs) write the hardware-specific code that corresponds to the display driver's platform-dependent driver (PDD) layer. Display driver writers can minimize the amount of time and effort that is needed to create a display driver by customizing one of the sample drivers provided in the Platform Builder. The display drivers in the Platform Builder represent a variety of display hardware devices. All of the sample drivers take advantage of the GPE to provide default drawing with hardware-specific issues addressed in the driver samples.

Primary Display Driver

If a Windows CE–based platform includes a display, Windows CE loads the default, or primary, display driver when the system starts up. By default, GWES loads a display driver named Ddi.dll. If the optional **HKEY_LOCAL_MACHINE\System\GDI\Drivers\Display** registry key is present, GWES loads the display driver that is indicated by that key. If the platform's version of Windows CE is configured to include windowing, the Window Manager uses the frame buffer that is supported by the primary display driver for all on-screen drawing.

In order to perform a drawing operation, an application must first get a device context for a window or the desktop by calling the **CreateWindow** and **GetDC** functions. The resulting device context is the application's drawing surface. If the application's Windows CE–based platform does not include support for windowing, and thus has no Window Manager, the application must call the **CreateDC** function with the name of the primary display driver's .dll file to create a device context directly.

Secondary Display Drivers

In addition to the primary display, Windows CE–based platforms can support additional display devices. These additional controllers might be used, for example, to support a Video Graphics Adapter (VGA) output to a CRT screen. Display drivers for such additional display devices are called secondary display drivers. Secondary display drivers are loaded only when an application calls the **CreateDC** function with the name of the secondary display driver's .dll file. **CreateDC** returns a handle to a device context that is associated with the secondary display driver. An application can use this device context just like a device context associated with the primary display driver; all of the text and graphics APIs work with the device context. However, applications cannot use any Window Manager functions with the device context because the Window Manager was not involved in creating it. Thus, applications are responsible for rendering the entire display on the secondary display device. If you want to create the appearance of windows, menus, dialog boxes, scroll bars, and so on on the secondary display, your application must draw those items itself, using the text and graphics APIs. If you want to show the contents of the primary display on the secondary display, your application can copy the frame buffer for the primary display to the frame buffer for the secondary display. The secondary display driver is unloaded when there are no remaining device contexts associated with it. When multiple display devices are being used, the GDI makes sure that all drawing calls are routed to the appropriate display driver.

Windows CE–based platforms can provide a built-in secondary display controller. The controller must be managed by a secondary display driver. The secondary display driver must be implemented as an ordinary Windows CE display driver, and expose the same display DDI as other display drivers.

If the display controller resides on a removable medium, such as a PC Card, the secondary display driver can be implemented as two DLLs or as a single DLL. In practice, most device driver writers choose to implement two separate DLLs: a secondary display driver that exposes the native display DDI and a stream driver that exposes the 11 stream interfaces. The Device Manager loads the stream interface driver when a user connects the display adapter to the system and registers the driver's special device file name; for example, "VGA1:". The secondary display driver is loaded when an application calls **CreateDC** and passes in the name of the secondary display driver's .dll file. The secondary display driver gets a handle to the stream interface driver by calling the **FileOpen** function on the "VGA1:" special device file. The secondary display driver handles all of the graphics processing. When it needs to communicate with the display controller, it uses the stream interface driver's **IOControl** function.

You may choose to create a single DLL that acts as both secondary display driver and stream driver instead. In this case, the Device Manager loads this combined driver in its own process space to serve as an ordinary stream interface driver, whereas GWES would load the same driver in its process space to act as the secondary display driver. If you choose to implement a combined driver, be aware that because the driver DLL is loaded in separate process spaces, it cannot share global data without using shared memory or a memory-mapped file. Furthermore, having the driver loaded twice wastes system resources.

Many of the stream interface functions are not used by display device drivers because display devices are not particularly oriented to working with streams of data. Display drivers need only have stubs for those functions. For these interfaces, the driver should handle the call and return. The driver's *XXX*_**Init** and *XXX*_**Deinit** functions are exceptions. The driver should handle calls to these functions correctly, as is described in "Developing Stream Interface Device Drivers." The Device Manager calls *XXX*_**Init** when the display adapter is connected to the system. When the adapter is disconnected, the Device Manager calls *XXX*_**Deinit**. *XXX*_**Deinit** deletes any data structures and registry entries that were created by the driver's *XXX*_**Init** function.

Like all Personal Computer Memory Card International Association (PCMCIA)–based drivers, the stream interface driver must create registry entries that enable the device to be detected. For more information, see "Registry Keys for Display Drivers."

Using Secondary Display Drivers with Pocket PowerPoint

The Microsoft Pocket PowerPoint application can use secondary displays if the display driver maintains certain registry keys when it is loaded and unloaded. At initialization, the driver should create a subkey for itself within the **HKEY_LOCAL_MACHINE\Drivers\Display\Active** key. This key is used to specify the secondary display driver's DLL name, as well as the driver's buffer and tap information. For more information, see "Registry Keys for Display Drivers." For removable display adapters, such as PC Card–based adapters, this initialization should take place when the Device Manager calls the display driver's *XXX*_**Init** function.

Pocket PowerPoint loads the display DDI library for the secondary display by calling the **CreateDC** function with the display DDI library name for the secondary display. For example, if the **Drivers\Display\Active\Voyager\Dll** subkey has the value **PCARDVGA.DLL**, Pocket PowerPoint calls **CreateDC** and passes the string PCARDVGA.DLL as an input parameter.

After the driver loads and initializes, Pocket PowerPoint calls the **GetDeviceCaps** function to identify the display device's resolution and color depth. The secondary display driver typically stores this information internally. When the driver supports multiple resolutions and those resolutions are selected by a separate application, such as a Control Panel application, this information may optionally be kept in the registry.

When the secondary display driver is unloaded, it should remove the **Drivers\Display\Active** subkey that it created during initialization. This indicates that the driver is no longer available. For removable display adapters, this de-initialization should take place when the Device Manager calls the drivers *XXX*_**Deinit** function after a user disconnects the display adapter from the system.

DDI Functions

All Windows CE–based display drivers must implement the DDI functions listed in the following table. However, only **DrvEnableDriver** must be exported from the display driver's DLL, which means that only **DrvEnableDriver** must bear this exact name. You can customize the names for the other functions because they are exposed to the GDI through function pointers that are returned by **DrvEnableDriver**. Of course, no matter what names you give these functions, they must follow the prototypes in the Winddi.h header file.

Function	Description
DrvAnyBlt	Bit block transfer, with stretching or transparency.
DrvBitBlt	General bit block transfer, with clipping and masking.
DrvContrastControl	Enables software adjustment of the display hardware's contrast.
DrvCopyBits	Sends a GDI-created print band to a printer driver.
DrvCreateDeviceBitmap	Creates and manages bitmaps.
DrvDeleteDeviceBitmap	Deletes a device bitmap.
DrvDisableDriver	Notifies the display driver that the GDI no longer needs it and is ready to unload the driver.
DrvDisablePDEV	Notifies the driver that the GDI no longer needs a particular display device.
DrvDisableSurface	Notifies the driver that the GDI no longer needs a particular drawing surface.
DrvEnableDriver	The initial entry point that is exposed by the driver, which returns pointers to the other DDI functions to the GDI.
DrvEnablePDEV	Returns a **PDEV** structure, which is a logical representation of a physical display device, to the GDI.
DrvEnableSurface	Creates a drawing surface and associates it with a **PDEV**.
DrvEndDoc	Sends any control information that is needed to finish printing a document.
DrvEscape	Used for retrieving information from a device that is not available in a device-independent DDI. This function is the same as in Windows NT, except that Windows CE does not support the **DrvDrawEscape** function.
DrvFillPath	Fills a drawing path with a brush.

Function	Description
DrvGetMasks	Gets the color masks for the display device's current mode.
DrvGetModes	Lists the display modes that are supported by the display device.
DrvMovePointer	Moves the pointer with a guarantee that the GDI will not interfere with the operation.
DrvPaint	Paints a specified region with a brush.
DrvPowerHandler	Called to handle POWER_UP and POWER_DOWN notifications.
DrvQueryFont	Gets font metric information.
DrvRealizeBrush	Creates a brush with parameters that are specified by the GDI.
DrvRealizeColor	Maps an RGB color onto the closest available color that is supported by the device.
DrvSetPalette	Sets the display device's palette.
DrvSetPointerShape	Sets the pointer to a new shape and updates the display.
DrvStartDoc	Sends any control information that is needed to start printing document.
DrvStartPage	Sends any control information that is needed to start printing a new page.
DrvStrokePath	Renders a drawing path.
DrvTransparentBlt	Bit block transfer, with transparency.
DrvUnrealizeColor	Maps a color in the display device's format onto an RGB value.

Using the GPE Classes

All sample display drivers included with the Platform Builder use the GPE classes. Although the GPE classes are optional, using them greatly facilitates the process of writing display drivers. If you use the GPE classes, you need only provide the new code that is necessary to make your display hardware operate correctly and perform acceleration.

The GPE classes require that your display hardware use a flat-frame buffer; that is, the display's memory must lie in a contiguous memory range. Modifying the GPE classes to use a non-contiguous frame buffer would require significant effort.

► **To create a display driver based on the GPE classes**

1. Create a directory for your project.
2. Copy the files from one of the sample driver directories, such as the S3Trio64 directory, to your project directory.

3. Replace all device-specific names in those files, such as S3Trio64, with your device's name.
4. Change the Config.cpp file so that it places your display device in a linear frame-buffer mode.
5. Disable all hardware-specific acceleration.
6. Build and test this non-accelerated driver.
7. Add your own hardware acceleration code.

For more information about GPE classes and methods, see the Microsoft Windows CE API Reference.

GDI Support Services for Display Drivers

The Windows CE GDI provides some services to support display drivers as predefined structures. The structures interact both with associated functions and with a few stand-alone C functions. Predefined structures provide support for brushes, palettes, translations, clipping regions, and stroke and fill paths. Stand-alone C functions provide support for device bitmaps and surfaces. The following table shows the structures and functions.

Structure or function	Description
BRUSHOBJ	Structure that represents a brush that is used for solid or patterned stroke and fill operations
BRUSHOBJ_pvAllocRbrush	Function that allocates memory for a brush
BRUSHOBJ_pvGetRbrush	Function that retrieves a pointer to the specified brush
CLIPOBJ	Structure that represents a clipping region
CLIPOBJ_bEnum	Function that enumerates clipping rectangles from a clipping region
CLIPOBJ_cEnumStart	Function that sets parameters for enumerating the rectangles in a clipping region
EngCreateDeviceBitmap	Function that causes the GDI to create a handle for a device bitmap
EngCreateDeviceSurface	Function that causes the GDI to create a device surface that the display driver manages
EngDeleteSurface	Function that informs the GDI that a device surface no longer is needed by the display driver
PALOBJ_cGetColors	Function that copies colors into a palette
PATHDATA	Structure that stores portions of a drawing path
PATHOBJ_bEnum	Function that enumerates **PATHDATA** records from a drawing path
PATHOBJ_vEnumStart	Function that readies a drawing path to have its

Structure or function	Description
	component line segments enumerated
PATHOBJ_vGetBounds	Function that returns the bounding rectangle for a drawing path
XLATEOBJ	Structure used in translating colors from one palette to another
XLATEOBJ_cGetPalette	Function that retrieves colors from an indexed palette

Accelerating Bit Block Transfers and Line Drawing

Display drivers do much of their work by means of a few basic operations. Thus, if you make those operations faster, you greatly improve the overall performance of your display driver. The operations that account for the bulk of a display driver's work are *bit block transfers* (blits) and line drawing. Any operation that transfers a rectangular group of pixels from main memory to display memory is a blit. Such operations include drawing rectangles that are filled with solid colors, displaying icons, and displaying cursors. Line drawing, for the purposes of acceleration, is limited to drawing straight lines.

These types of acceleration can be done either in hardware or in software. Hardware acceleration for blits and line drawing is generally faster but is not available in all display hardware. However, even if your display hardware cannot accelerate the functions, you may still be able to achieve better performance than the default provided by the GPE classes. For example, you can write routines that take advantage of the specific characteristics of your display hardware to perform blits and draw lines as efficiently as possible.

The capabilities of display hardware vary considerably. Simple display hardware requires that every pixel of the display be set by the display driver. Such hardware cannot accelerate blits or line drawing. Display hardware that is more complex can effectively complete those tasks much faster than a display driver can.

The following sections describe how to support both hardware and software acceleration of blit and line drawing operations in your display driver. The Platform Builder includes sample display drivers that illustrate both hardware and software acceleration.

Accelerating Bit Block Transfers

With Windows CE 2.0 and later, a display driver can use up to three levels of blit processing: the default blit emulation that is provided by the GPE, the software acceleration that is provided by the emulation library or by your own custom code, and the hardware acceleration that is supported by the display device hardware. The display driver can choose, based on the parameters of individual operations, which method to use for each blit that it performs.

The Platform Builder provides the GPE library, which is located under Public\Common\OAK\Lib. This class library, offered in binary form, serves as the foundation for Windows CE–based display drivers. The GPE supplies default processing for blits in its **EmulatedBlt** function.

Emulation Library for Software-Accelerated Blits

Windows CE provides sample code for creating an emulation library for software-accelerated blits. The sample code is located in the Public\Common\OAK\Drivers\Display\Emul directory. The library contains emulated blit functions with destination pixel depths of 2, 8, and 16 bits per pixel. You can use the emulated blits to improve performance over the default blit processing that is provided in the GPE. Of particular interest to embedded system developers is that the source code can serve as a template for writing additional software-accelerated blits that are tailored to the display hardware.

Sample Blit Acceleration

By default, a display driver routes all blits to the GPE. As alternatives, a display driver can route blit handling to the emulation library or directly to the hardware.

The S3Virge display driver serves as a model to demonstrate how a display driver can invoke all three of these methods. The S3Virge sample code is located in the Platform\CEPC\Drivers\Display\S3Virge directory.

The following code example is drawn from the sample S3Virge display driver contained in the Platform Builder. It shows how blit processing begins when the GDI calls the driver's **BltPrepare** function. The driver initializes the blit parameters and determines which function to use to perform the individual blits. Typically, the driver initializes the GPE to handle default blit processing. The following code example shows this initialization.

```
SCODE
S3Virge::BltPrepare(GPEBltParms *pBltParms)
{
    // Put debug messages and optional timing processing here.

    pBltParms->pBlt = EmulatedBlt; // Generic BLT processing
```

For improved performance, **BltPrepare** can examine the characteristics of the blit and the associated display surfaces to determine whether an accelerated blit is appropriate. After the initialization code, the display driver can contain code for hardware or software accelerations. The sample driver includes or excludes this code at compilation time. The **ENABLE_ACCELERATION** preprocessor directive specifies inclusion of the code for hardware accelerations. The **ENABLE_EMULATION** directive specifies inclusion of the code for software-accelerated emulation.

After setting the default handler, the S3Virge driver dispatches blits to hardware acceleration, if available, and then to software-accelerated emulation, if available. The driver defines a macro that simplifies calls to functions in the emulation library. Next, it checks whether the destination surface is in video memory. This is an important check that most display drivers must make before using hardware acceleration, because the GDI uses the display driver to render printer output, as well as display output. Printer output is rendered in system memory, and not in video memory. Most display hardware is able to perform accelerated drawing in only two situations; when the destination surface is in video memory, for blits without a source surface; or when both the source and destination surfaces are in video memory, for blits with a source surface. Therefore, if the hardware has this limitation, the driver must check the source and destination surfaces before calling the acceleration function.

The following code example shows how the driver evaluates the raster operation (ROP) code and directs the blit to a supported hardware acceleration, when available. The SRCCOPY ROP illustrates how the driver checks for a source surface in video memory before invoking hardware acceleration.

```
#ifdef ENABLE_ACCELERATION

#define FUNCNAME(basename) (SCODE (GPE::*)(struct GPEBltParms
*))Emulator::Emulated##basename

    if ( pBltParms->pDst->InVideoMemory() ) {

switch( pBltParms->rop4 )
    {
        case 0x0000:// BLACKNESS
            pBltParms->solidColor = 0x0;
            pBltParms->pBlt = (SCODE (GPE::*)
(struct GPEBltParms *)) AcceleratedFillRect;
            break;
        case 0xFFFF:// WHITENESS
            pBltParms->solidColor = 0xffffff;
            pBltParms->pBlt = (SCODE (GPE::*)
(struct GPEBltParms *))AcceleratedFillRect;
            break;
        case 0xF0F0:// PATCOPY
        case 0x5A5A:// PATINVERT
            if (pBltParms->pLookup || pBltParms->pConvert)   // Emulate
// Color Convertion
            {
                RETAILMSG(VIRGE_VERBOSE_MSGS, (TEXT("S3Virge::Blt -
Emulate Color Conversion\r\n")));
                break;
            }
            if (pBltParms->solidColor == -1)
            {
                RETAILMSG(VIRGE_VERBOSE_MSGS, (TEXT("S3Virge::Blt -
PATCOPY - patterned brush\r\n")));
                DEBUGWAIT(GPE_ZONE_BLT_LO)
                pBltParms->pBlt = (SCODE (GPE::*) (struct GPEBltParms
*)) AcceleratedPatCopyBlt;
            }
            else
            {
                RETAILMSG(VIRGE_VERBOSE_MSGS, (TEXT("S3Virge::Blt -
PATCOPY - solid brush\r\n")));
                DEBUGWAIT(GPE_ZONE_BLT_LO)
                pBltParms->pBlt = (SCODE (GPE::*) (struct GPEBltParms
*)) AcceleratedFillRect;
            }
            break;
        case 0x6666:// SRCINVERT
        case 0x8888:// SRCAND
        case 0xCCCC:// SRCCOPY
        case 0xEEEE:// SRCPAINT
            // Cannot accelerate if src not in video memory
            // Cannot accelerate color translations
            if (!pBltParms->pSrc->InVideoMemory() || pBltParms->pLookup
|| pBltParms->pConvert)
                break;

            if (pBltParms->bltFlags & BLT_STRETCH)
            {
                // Stretch blit
                RETAILMSG(VIRGE_VERBOSE_MSGS,(TEXT("Stretch
Blt!\r\n")));
                break;
            }
            pBltParms->pBlt = (SCODE (GPE::*) (struct GPEBltParms *))
AcceleratedSrcCopyBlt;
            break;
```

The following code example for the S3Virge display driver shows how to use the blit emulation library. After setting the default blit handler and handling hardware accelerations, the driver checks for possible software accelerations. When the accelerations are available, it sets the blit function pointer to the corresponding function in the emulation library.

Before invoking a blit function in the emulation library, the driver must check for conditions that would prevent the blit from using emulation. This is critical, because the functions in the emulation library require that the driver validate the blit before calling an emulation function. In the previous example, the driver checked for a solid brush because the emulation library does not support pattern brushes. The emulation library also cannot handle color conversions, bit depth conversions, stretching, and transparency. The following code example shows how the driver checks that none of these restrictions apply.

```
if (pBltParms->pBlt == EmulatedBlt) {
        //
        // Cancel any parameter values that would make this
        // blit non-emulatable.
        //
        if (!(EGPEFormatToBpp[pBltParms->pDst->Format()] != 8        ||
             (pBltParms->bltFlags & (BLT_TRANSPARENT | BLT_STRETCH)) ||
             pBltParms->pLookup                                      ||
             pBltParms->pConvert))
```

After confirming that the emulation library functions can perform a blit, the display driver examines the ROPs, dispatching supported ROPs to the appropriate function. For simplicity, the driver specifies the ROP4 value. For masking blits, the entire **WORD** is relevant. For non-masking blits, only the least significant byte—the ROP3 value—is relevant. ROPs are defined as **DWORD**s for compatibility with Windows-based desktop platforms, although Windows CE–based display drivers use only the most significant **WORD** of a ROP value. Windows CE–based display drivers ignore the lower **WORD**, which contains ROP compiler directives. The following code example shows how the display driver checks the ROP4 value to choose which blit function to use.

```
switch (pBltParms->rop4)
    {
        case 0x0000:    // BLACKNESS
            pBltParms->solidColor = 0;
            pBltParms->pBlt = FUNCNAME(BltFill08);
break;
        case 0xFFFF:    // WHITENESS
            pBltParms->solidColor = 0x00ffffff;
            pBltParms->pBlt = FUNCNAME(BltFill08);
break;
```

```
        case 0xF0F0:    // PATCOPY
            if( pBltParms->solidColor != -1 )
            {
                pBltParms->pBlt = FUNCNAME(BltFill08);
            }
            break;
```

Additional Accelerated Sample Drivers

Like the S3Virge display driver, other display drivers provided in the Platform Builder demonstrate how to invoke hardware accelerations and how to make use of the emulation library.

The S3Trio64 driver is in the Platform\CEPC\Drivers\Display\S3Trio64 directory. The S3Trio64 driver demonstrates support for hardware accelerations and use of the emulation library. The driver also supports blit handling for destination surfaces that are both 8 bits per pixel (bpp) and 16 bpp. For example, in the case of a mask PATCOPY text blit with a mask that is 1 bpp, the driver uses the emulation library's **BltText16** function to handle the blit. With a mask that is 4-bpp, the driver uses the emulation library's **BltalphaText16** function.

```
if (pBltParms->pBlt == EmulatedBlt) {
#ifdef FB16BPP
    switch (pBltParms->rop4)
    {
        case 0xAAF0:
// Special PATCOPY ROP for text output--fill where mask is set.
            // If the brush is not a pattern brush
            if( (pBltParms->solidColor != -1) &&
                (pBltParms->pDst->Format() == gpe16Bpp) )
            {
                WaitForNotBusy();
                if (pBltParms->pMask->Format() == gpe1Bpp)
                {
                    pBltParms->pBlt =
FUNCNAME(BltText16);
                    return S_OK;
                }
                else   // Antialiased text
                {
                    pBltParms->pBlt =
FUNCNAME(BltAlphaText16);
                    return S_OK;
                }
            }
            break;
        default:  // Other ROP4s
            ;
    }
```

For hardware accelerations and use of the emulation library, see the Citizen driver in the Platform\ODO\Drivers\Display\Citizen directory.

Accelerating Line Drawing

Windows CE 2.0 and later includes support for accelerating line drawing. The levels of accelerated line drawing are largely the same as for accelerated blit operations. The discussion of accelerating bit block transfers in the previous sections is relevant for line drawing as well, except that the emulation library provided in the Platform Builder does not provide software-accelerated line drawing. Developers can add software-accelerated line drawing functions, if necessary.

Display drivers route line drawing to the GPE, to custom software acceleration routines, or directly to the hardware. The following code example from the S3Virge display driver sample shows how these methods are invoked. This source code is very similar in design to the acceleration processing performed for blit operations.

```
SCODE
S3Virge::Line(GPELineParms *pLineParms, EGPEPhase phase)
{

#ifdef ENABLE_ACCELERATION
    if (phase == gpeSingle || phase == gpePrepare)
    {
        DispPerfStart (ROP_LINE);
        pLineParms->pLine = EmulatedLine;
        if (pLineParms->pDst->InVideoMemory() && pLineParms->mix ==
0x0d0d)
        {
#if VIRGE_VERBOSE_MSGS
            switch (phase)
            {
            case gpeSingle:
                RETAILMSG(VIRGE_VERBOSE_MSGS, (TEXT("in single\n\r")));
                break;
            case gpePrepare:
                RETAILMSG(VIRGE_VERBOSE_MSGS, (TEXT("in prepare\n\r")));
                break;
            }
#endif
            SelectSolidColor(pLineParms->solidColor);
            pLineParms->pLine = (SCODE (GPE::*)(struct GPELineParms *))
AcceleratedSolidLine;
        }
    } else if (phase == gpeComplete) {
        DispPerfEnd (0);
    }
#else
    pLineParms->pLine = EmulatedLine;
#endif
    return S_OK;
}
```

The S3Virge driver conditionally compiles the hardware acceleration code with the **ENABLE_ACCELERATION** preprocessor directive. Line drawing begins with a call to the driver's **Line** function. For improved performance, the driver's **Line** function can examine the characteristics of the line drawing and the associated surfaces to determine whether an accelerated form of line drawing is appropriate. The default line drawing function is set to the **EmulatedLine** function of the GPE. The driver checks for a destination surface in video memory and checks the line drawing parameters. If the parameters are valid for acceleration, the line drawing function is set to the driver's **AcceleratedSolidLine** function.

Functions in the Emulation Library

The emulation library includes the following functions. Developers can add additional functions, if necessary.

EmulatedBltAlphaText02

Special-case fast blit function for rendering antialiased text. This function assumes a mask surface containing the 4-bpp alpha bitmap for the glyph.

File name	ROP	Source bit depth	Target bit depth
Ebalph02.cpp	AAF0	–	02

EmulatedBltAlphaText16

Special-case fast blit function for rendering antialiased text. This function assumes a mask surface containing the 4-bpp alpha bitmap for the glyph.

File name	ROP	Source bit depth	Target bit depth
Ebalph16.cpp	AAF0	–	16

EmulatedBltSrcCopy0202

Implements blit(SRCCOPY)

File name	ROP	Source bit depth	Target bit depth
Ebcopy02.cpp	CCCC	02	02

EmulatedBltSrcCopy0808

Implements blit(SRCCOPY)

File name	ROP	Source bit depth	Target bit depth
Ebcopy08.cpp	CCCC	08	08

EmulatedBltSrcCopy1616

Implements blit(SRCCOPY)

File name	ROP	Source bit depth	Target bit depth
Ebcopy16.cpp	CCCC	16	16

EmulatedBltDstInvert02

Implements Patblt(DSTINVERT)

File name	ROP	Source bit depth	Target bit depth
Ebdinv02.cpp	5555	–	02

EmulatedBltDstInvert08

Implements Patblt(DSTINVERT)

File name	ROP	Source bit depth	Target bit depth
Ebdinv08.cpp	5555	–	08

EmulatedBltFill02

Implements Patblt(PATCOPY) for ROP F0F0, Patblt(BLACKNESS) for ROP 0000, and Patblt(WHITENESS) for ROP FFFF

File name	ROP	Source bit depth	Target bit depth
Ebfill02.cpp	0000, FFFF, F0F0	–	02

EmulatedBltFill08

Implements Patblt(PATCOPY) for ROP F0F0, Patblt(BLACKNESS) for ROP 0000, and Patblt(WHITENESS) for ROP FFFF

File name	ROP	Source bit depth	Target bit depth
Ebfill08.cpp	0000, FFFF, F0F0	–	08

EmulatedBltFill16

Implements Patblt(PATCOPY) for ROP F0F0, Patblt(BLACKNESS) for ROP 0000, and Patblt(WHITENESS) for ROP FFFF

File name	ROP	Source bit depth	Target bit depth
Ebfill16.cpp	0000, FFFF, F0F0	–	16

EmulatedBltPatInvert02

Implements Patblt(PATINVERT)

File name	ROP	Source bit depth	Target bit depth
Ebpinv02.cpp	5A5A	–	02

EmulatedBltPatInvert08

Implements Patblt(PATINVERT)

File name	ROP	Source bit depth	Target bit depth
Ebpinv08.cpp	5A5A	–	08

EmulatedBltSrcAnd0202
Implements blit(SRCAND)

File name	ROP	Source bit depth	Target bit depth
Ebsand02.cpp	8888	02	02

EmulatedBltSrcAnd0808
Implements blit(SRCAND)

File name	ROP	Source bit depth	Target bit depth
Ebsand08.cpp	8888	08	08

EmulatedBltSrcAnd1616
Implements blit(SRCAND)

File name	ROP	Source bit depth	Target bit depth
Ebsand16.cpp	8888	16	16

EmulatedBltSrcInvert0202
Implements blit(SRCINVERT)

File name	ROP	Source bit depth	Target bit depth
Ebsinv02.cpp	6666	02	02

EmulatedBltSrcInvert0808
Implements blit(SRCINVERT)

File name	ROP	Source bit depth	Target bit depth
Ebsinv08.cpp	6666	08	08

EmulatedBltSrcInvert1616
Implements blit(SRCINVERT)

File name	ROP	Source bit depth	Target bit depth
Ebsinv16.cpp	6666	16	16

EmulatedBltSrcPaint0202
Implements blit(SRCPAINT)

File name	ROP	Source bit depth	Target bit depth
Ebspnt02.cpp	EEEE	02	02

EmulatedBltSrcPaint0808
Implements blit(SRCPAINT)

File name	ROP	Source bit depth	Target bit depth
Ebspnt08.cpp	EEEE	08	08

EmulatedBltSrcPaint1616

Implements blit(SRCPAINT)

File name	ROP	Source bit depth	Target bit depth
Ebspnt16.cpp	EEEE	16	16

EmulatedBltText02

Special-case fast blit function for rendering solid-color filled text with a mask.

File name	ROP	Source bit depth	Target bit depth
Ebtext02.cpp	AAF0	–	02

EmulatedBltText08

Special-case fast blit function for rendering solid-color filled text with a mask.

File name	ROP	Source bit depth	Target bit depth
Ebtext08.cpp	AAF0	–	08

EmulatedBltText16

Special-case fast blit function for rendering solid-color filled text with a mask.

File name	ROP	Source bit depth	Target bit depth
Ebtext16.cpp	AAF0	–	16

Supporting Antialiased Fonts

Antialiased fonts work on any device whose display driver supports them. Typically, antialiasing is supported on devices that are not palettized. If the display driver supports antialiased fonts, it must notify the GDI of this capability at system startup. It does this by returning GCAPS_GRAY16 from the **GetGraphicsCaps** function. The S3Trio64 sample display driver demonstrates how to support antialiasing. The driver source code is located in the Platform\Cepc\Drivers\Display\S3trio64 directory.

Supporting ClearType

Windows CE versions 2.12 and later support the ClearType font technology that is provided by Microsoft. ClearType dramatically improves font sharpness on color LCD displays. If the display driver supports ClearType, it must notify the GDI of this capability at system startup by returning GCAPS_CLEARTYPE in the call to the **GetGraphicsCaps** function. The Citizen sample display driver demonstrates how to support ClearType. The driver source code is located in the Platform\Odo\Drivers\Display\Citizen directory.

The display driver invokes a software emulation to render a ClearType glyph. For example, in the Citizen driver, text output with a solid brush is usually routed to the **BltText08** emulation function. For the edge pixels on the glyph, the display function pointer is set to a special **BltCleartype** emulation for the 3-3-2 palette and the Halftone palette.

```
case 0xAAF0:    // Special PATCOPY ROP for text output--fill where mask
                //is 1.
                // Not a pattern brush?
        if( pBltParms->solidColor != -1 )
        {
            if (pBltParms->pMask->Format() == gpe1Bpp)
            {
                DEBUGMSG(GPE_ZONE_BLT_LO,(TEXT("CITIZEN::Blt -
0xAAF0\r\n")));
                pBltParms->pBlt = FUNCNAME(BltText08);
                return S_OK;
            }
            else if (pBltParms->pMask->Format() == gpe8Bpp)  //
ClearType
            {
                DEBUGMSG(GPE_ZONE_BLT_LO,(TEXT("CITIZEN::Blt ClearType -
0xAAF0\r\n")));
                pBltParms->pBlt = FUNCNAME(BltCleartypeHalftoneText08);
//              pBltParms->pBlt = FUNCNAME(BltCleartype332Text08);
                return S_OK;
            }
        }
        break;
```

Including Display Drivers in an OS Image

OEMs can include the emulation library and the GPE class library in their Windows CE OS image. The emulated blit functions are compiled and linked into a single library called Emul.lib. The display driver links to this library through a link directive in the driver's Sources file. For example, the Sources file for the S3Trio64 display driver includes Emul.lib in its list of target libraries. This links the driver with Emul.lib. The Platform Builder provides the GPE library in binary form. The GPE library is also included in the driver's Sources file.

The Platform.bib file in the Platform\Cepc\Files directory directs the RomImage tool, which is the tool that creates OS images for Windows CE, to include the appropriate driver in the OS image. See the Platform Builder for complete documentation on using the Romimage tool and creating OS images.

If you are building a platform from one of the sample Platform.bib files that is provided in the Platform Builder, you can change which display driver the Romimage tool puts into the OS image. The Platform.bib file in the Platform\Cepc\Files directory directs the Romimage tool to include the S3Trio64 driver in the OS image. Other display drivers may take the place of S364Trio. See Platform.bib for the list of environment variables that may be set to direct the Romimage tool to use a different display driver. The 2BPP driver is the default display driver for the Windows CE hardware reference platform. The Platform.bib file in the Platform\Odo\Files directory lists the environment variables that may be set to direct Romimage to use a different display driver.

Testing Display Drivers

The Platform Builder includes a sample application, VideoApp, that the display driver writer can use to test a display driver. The application tests the driver's ability to draw lines, polygons, rectangles, text, cursors, and to display various color patterns. The application source code is located in the Platform\Cepc\Test\Videoapp directory. The VideoApp.exe is located in the Platform\Cepc\Drivers\Display\Test directory.

Testing a new driver with the VideoApp application verifies that the driver correctly performs a number of common graphics operations. The application is not intended as an exhaustive test of all possible operations. The display driver writer is responsible for thorough testing of the driver.

Profiling Display Driver Performance

The Platform Builder includes a sample profiling tool, DispPerf, that the display driver writer can use to profile the performance of a display driver. DispPerf builds a table that lists, for each ROP code that is profiled, the frequency count, elapsed time in microseconds, and average elapsed time in microseconds, for ROPs handled by the default GPE emulation, by the software emulation library, and by hardware accelerations. To measure these times accurately, DispPerf can be used only on Windows CE–based platforms that support the **QueryPerformanceCounter** and **QueryPerformanceFrequency** functions with a recommended resolution of 1 microsecond.

The S3Virge driver demonstrates how to use the profiler for measuring performance of blits and line drawing. The display driver writer may extend the profiler to measure the performance of additional display functions. Source code for DispPerf is in the Platform\Cepc\Drivers\Display\S3virge\Dispperf directory.

The driver starts profiling of blit operations during the call to the **BltPrepare** function when it calls the **DispPerfStart** function. The following code example shows how the driver initializes its display function pointer to use the default emulation that is provided by the GPE.

```
SCODE
S3Virge::BltPrepare(GPEBltParms *pBltParms)
{

    DispPerfStart (pBltParms->rop4);

    pBltParms->pBlt = EmulatedBlt; // Catch all
```

When the driver is able to handle the ROP with hardware acceleration or software emulation, it changes its display function pointer appropriately. The driver also calls the **DispPerfType** function to record which type of acceleration is used to handle the ROP. For example, the following code example demonstrates how the driver calls **DispPerfType** after the driver successfully sets the display function pointer to a hardware-accelerated function.

```
// Performance Logging Type
    if (pBltParms->pBlt != EmulatedBlt) {
        DispPerfType(DISPPERF_ACCEL_HARDWARE);
    }
```

When the blit operation completes, the **BltComplete** function stops profiling with a call to the **DispPerfEnd** function.

The DO_DISPPERF environment variable controls whether the code to support profiling is compiled into the display driver. Refer to the Sources file for the S3Virge driver to see how the environment variable causes the correct compiler directives to be set.

DispPerf can be invoked from the Command Shell on a Windows CE–based platform, or it can be launched remotely from the Windows CE Debug Shell. The following example shows the command syntax for **DispPerf**.

```
dispperf [-c[w*]] [-d [> file]]
```

The *-c* option clears the profiler buffer. If the letter "w" appears one or more times following the *-c* option, **DispPerf** calls the **CreateWindow** function that number of times and profiles the resulting display driver operations. The *-d* option dumps the profiler buffer in tabular form. If the *-d* option is followed by (>) plus a file name, the output is written to the named file. Otherwise, it appears on the console where the command was issued.

The following code example shows how **DispPerf** clears its buffer of all profiling information.

```
Dispperf -cwwwwwd
```

It then calls the **CreateWindow** function five times, profiling the performance of the display functions. The following code example shows how **DispPerf** displays its table of profiling information to the console.

```
Dispperf -c
Pword stat_rpt.pwd
Dispperf -d > results.txt
```

In this example, **DispPerf** clears its buffer of all profiling information. The Microsoft® Pocket Word application is launched and opens the document Stat_rpt.pwd. **DispPerf** tracks the profiled display functions that are called by Pocket Word. Finally, **DispPerf** writes its table of profiling information to the Results.txt file.

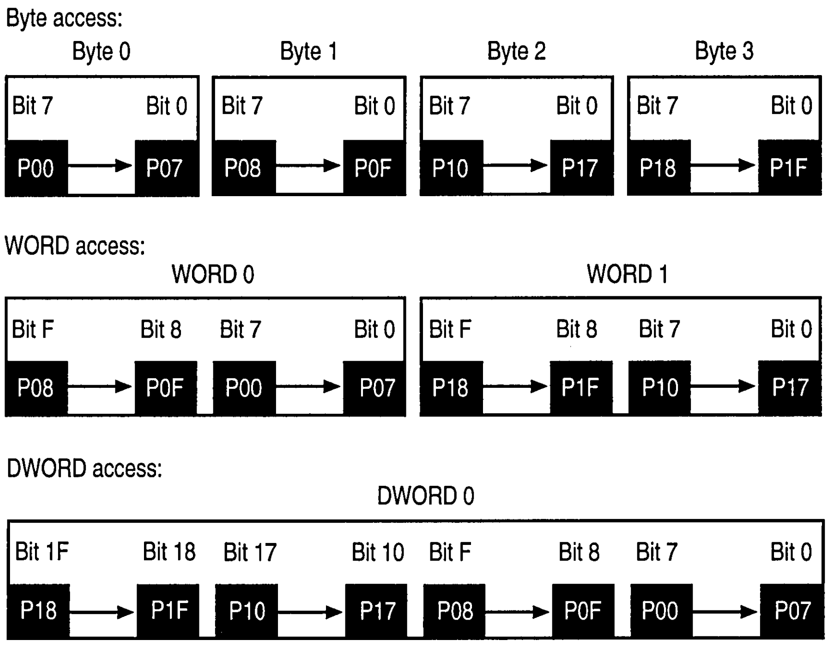
Display Buffer Formats

The Windows CE GDI supports displays with a wide variety of color depths and color models, ranging from 1-bit color to palettized color to true 32-bit red, green, and blue (RGB) color. Each format also supports several pixel orderings, depending on whether access to the display memory is by bytes, **WORD**s, or **DWORD**s.

For all display buffer formats, the order of pixels on the display is from left to right and from top to bottom. That is, pixel (0,0) appears at the upper-left corner of the display, and pixel (*width* –1, *height* –1) appears at the lower-right corner. For more information about the arrangement of video memory for each display buffer format that Windows CE supports, see the following sections.

Using 1 Bit per Pixel

The 1-bpp format is for simple black-and-white displays. Black is represented by 0, and white is represented by 1. Pixel (0,0) is packed into the highest-order bit of the first byte of display memory. The following illustration shows the arrangement of memory for the format.

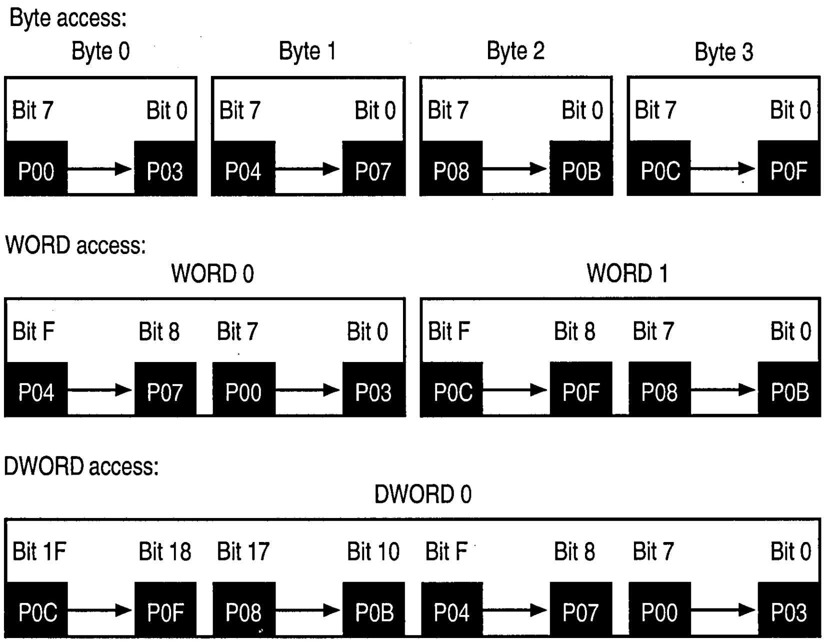


Using 2 Bits per Pixel

The 2-bpp format is typically used for 4-level grayscale displays, although any 4-entry palette works. The following table shows the bits for the associated gray levels.

Bit 1	Bit 0	Gray level
0	0	Black
0	1	Dark gray
1	0	Light gray
1	1	White

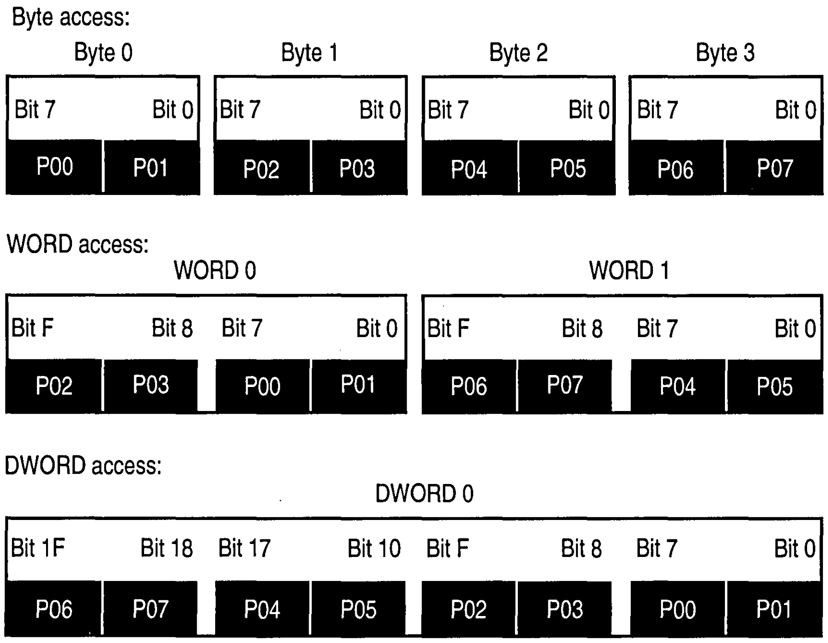
The following illustration shows the arrangement of memory for the format.



Using 4 Bits per Pixel

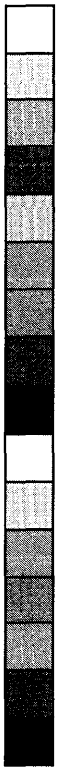
The 4-bpp format is usually a palettized format. The frame buffer itself can be implemented either as 2 pixels packed into each byte or as 1 pixel per byte.

The following illustration shows the arrangement of memory for the format.



If you choose to implement just 1 pixel per byte, the driver should represent the display mode as 8 bpp with a 16-color palette. The relevant bits in each byte should be the 4 lowest bits; the 4 highest bits should always be 0. For more information on formats in which single pixels do not fill entire bytes, see "Using 5 or 6 Bits per Pixel."

The following illustration shows the standard Windows CE 16-color palette, which you should use to obtain the best results.



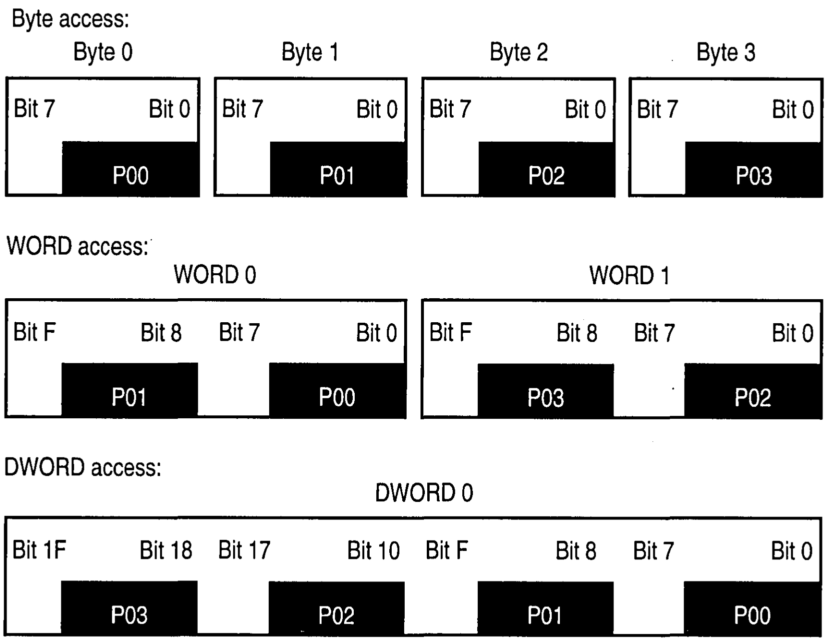
Color:	Red	Green	Blue
White	255	255	255
Teal	0	255	255
Purple	255	0	255
Blue	0	0	255
Light gray	192	192	192
Dark gray	128	128	128
Dark teal	0	128	128
Dark purple	128	0	128
Dark blue	0	0	128
Yellow	255	255	0
Green	0	255	0
Dark yellow	128	128	0
Dark green	0	128	0
Red	255	0	0
Dark red	128	0	0
Black	0	0	0

Using 5 or 6 Bits per Pixel

The 5-bpp or 6-bpp format should always use a whole byte for each pixel. The relevant bits must be the low-order bits in the pixel, with unused high-order bits containing 0s.

With displays that use 5 bpp or 6 bpp, you can choose either to palettize the colors or to use the bits in the pixel to represent the colors directly. For example, you can make 6 bpp a 64-entry palettized display, or you can use the 6 bits in each pixel to represent 4 levels each of red, green, and blue directly.

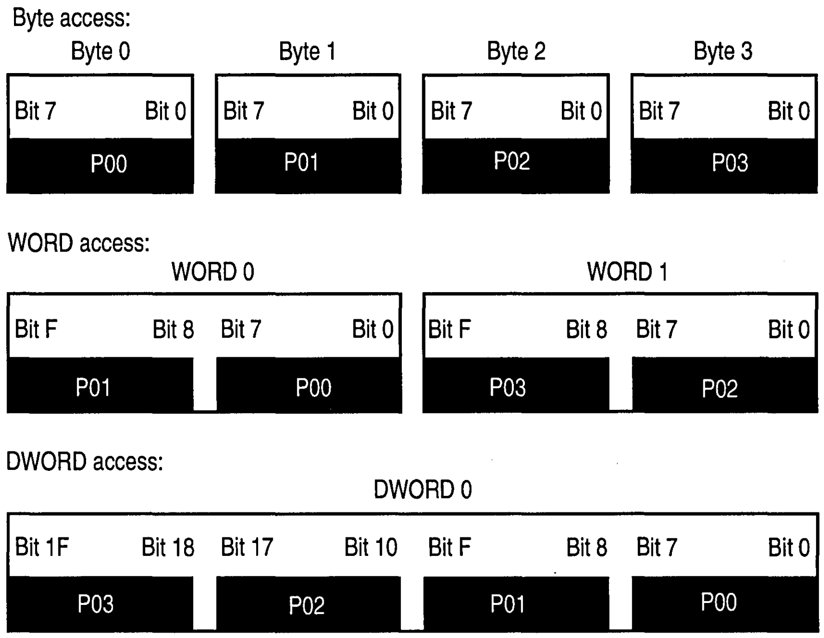
The following illustration shows the arrangement of memory for the format.



Using 8 Bits per Pixel

The 8-bpp format ideally should use a software-changeable palette that maps 8-bit values onto 24-bit colors. For better performance, compatibility, and image quality, Microsoft recommends using a palette that contains the default Windows CE palette, although this is not required.

The following illustration shows the arrangement of memory for the format.



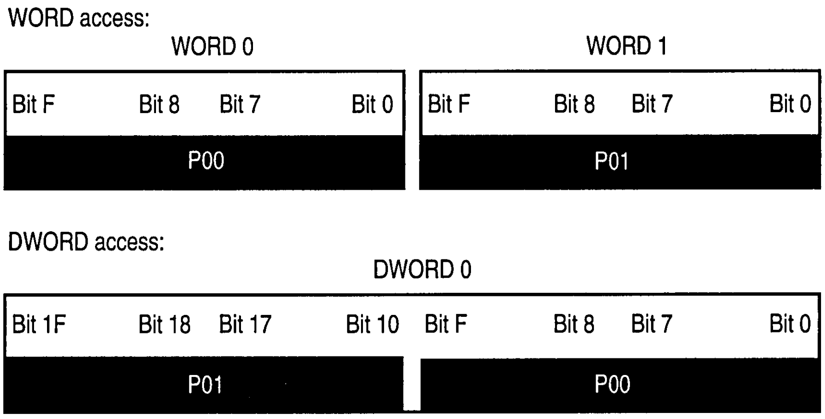
Using 15 or 16 Bits per Pixel

The 15-bpp or 16-bpp format is a masked, non-palettized format. For either 15 bpp or 16 bpp, 1 pixel is stored in each 2-byte **WORD**. The 15-bpp format wastes the high-order bit of each word. The following table shows the masks that Microsoft recommends to extract red, green, and blue values.

Color	15-bit mask	16-bit mask
Red	0x7C00	0xF800
Green	0x3E00	0x07E0
Blue	0x001F	0x001F

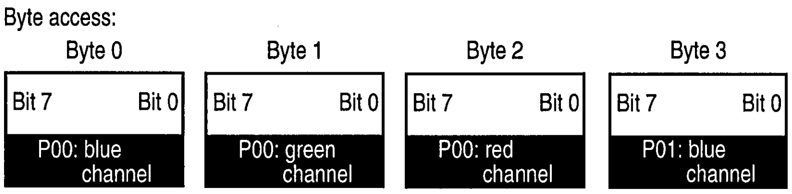
As the masks show for 15 bpp, the low-order 15 bits of each word contain the pixel data. The unused bit should contain 0.

The following illustration shows the arrangement of memory for the format.



Using 24 Bits per Pixel

The 24-bpp format is a true-color format, in which each pixel stores 8 bits for red, green, and blue. The advantage of this format is that image quality is very good. Because each pixel occupies exactly 3 bytes, the pixels can be packed together without wasting memory. The drawback to this format is that because half the pixels in this scheme cross **DWORD** boundaries, there is a performance penalty in accessing and decoding pixels. The following illustration shows the arrangement of memory for the format.

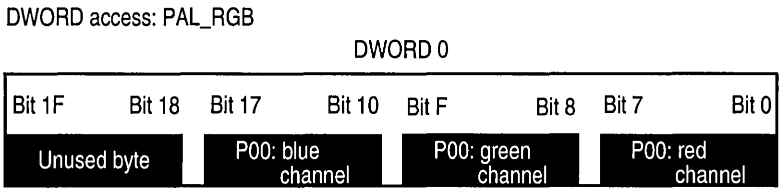


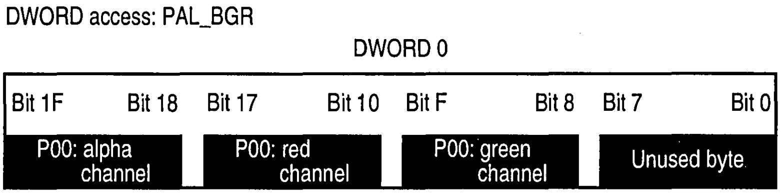
Using 32 Bits per Pixel

Like the 24-bpp format, the 32-bpp format is a true-color format. Unlike the 24-bpp format, however, the 32-bpp format does not cause pixels to cross **DWORD** boundaries, although this format is less efficient in memory use. There are two ways to arrange the color channels in this format. One method puts blue in the least significant byte of each pixel, and the other method puts red in the least significant byte. These options correspond to the PAL_BGR and PAL_RGB modes. Microsoft recommends using PAL_RGB for slightly better performance. The following table shows the masks that you can use to extract red, green, blue, and alpha channels from each pixel.

Color	**PAL_RGB mask**	**PAL_BGR mask**
Red	0x000000FF	0x00FF0000
Green	0x0000FF00	0x0000FF00
Blue	0x00FF0000	0x000000FF

If your product does not use an alpha channel, use 0x00000000 as the mask for alpha. The following illustration shows the arrangement of memory for the format.





Display Hardware Recommendations

For display hardware used with Windows CE, Microsoft makes several recommendations to improve performance and to facilitate driver development. Following the recommendations enables you to implement your display driver more easily, although you can write a fully functional driver even if your hardware does not conform to these recommendations.

Microsoft strongly recommends that your display hardware uses a linear-frame buffer. All the display's memory should be contiguous, and one linear-access window should cover the entire frame buffer. If your hardware does not meet these qualifications, you must make substantive modifications to the GPE classes, if you choose to use them. For more information, see "Using the GPE Classes."

Your display hardware should use a supported combination of pixel format, packing, and pixel ordering. For more information, see "Display Buffer Formats." The display hardware's frame buffer should have the following properties:

- Top-down format, with pixel (0,0) at the top left and pixel (*width* –1, *height* –1) at the lower right.
- The frame buffer's stride—the number of bytes in memory that it takes to represent one scan line on the display—should be a multiple of 4 bytes, even if this means padding the end of each scan line with unused bytes.
- The entire frame buffer must be accessible by the CPU without requiring the CPU to perform bank selection.
- Frame buffers should not use bit planes, in which separate frame buffers are used for each color channel or intensity component.

Microsoft also recommends using display hardware that can accelerate the following operations, in order of decreasing importance:

- Solid-color fills, specifically, blit operations whose *pbo->iSolidColor* member is not 0xFFFFFFFF.
- SRCCOPY blit operations.
- Cursor display, if your platform uses a cursor.
- Solid-line drawing with subpixel precision.
- Masked SRCCOPY blit operations.

Registry Keys for Display Drivers

When a Windows CE–based platform contains a display driver, it is automatically loaded by GWES at system startup. By default, GWES loads a driver named Ddi.dll. To change the name of the default display driver, use the **HKEY_LOCAL_MACHINE\System\GDI\Drivers\Display** key to override the default display driver DLL name. The registry key should be placed in your Platform.reg file.

```
[HKEY_LOCAL_MACHINE\System\GDI\Drivers]
    Display="DDI.DLL"
```

The device driver writer can optionally include a registry key that provides additional information about the driver. This information is not used by the GDI, but may be useful to application. It may also be useful to the driver when it supports, for example, multiple resolutions, which a user can select from a Control Panel application that is provided by the display driver writer. In this case, the driver could examine the registry to determine which resolution to set when the display driver loads. The following example shows a registry key for a native display driver that supports a Color Graphix Voyager PCMCIA-based display device. The key stores the native driver's DLL name, screen size, and color depth.

```
[HKEY_LOCAL_MACHINE\Drivers\Display\PCARDVGA]
    Dll="PCARDVGA.DLL"
    CxScreen=Dword:280
    CyScreen=Dword:F0
    Bpp=Dword:8
```

Registry Keys for Removable Display Adapters

When a secondary display adapter is provided on removable hardware, such as a PC Card, the display driver is typically implemented as two drivers: a native driver that exposes the native DDI and a stream interface driver that exposes the stream interfaces. The stream driver must initialize the registry with the keys that allow it to be detected and loaded. When the PC Card is inserted, the Device Manager initiates a detection sequence to determine which driver should service the card. If the PC Card has a Plug and Play identifier, it is used to determine which driver to load. Otherwise, the Device Manager tries the detection functions for all the PC Card drivers that are installed on the system until one of them reports that it recognizes the PC Card. Once the correct driver is located, the Device Manager loads the driver whose name is provided as the registry key along with the detection function that succeeded in detecting the PC Card. The following example shows how the Device Manager then loads the driver, registers its special device file name, and calls the driver's *XXX*_**Init** function.

```
[HKEY_LOCAL_MACHINE\Drivers\PCMCIA\VoyagerVGA]
    Dll="VOYAGER.DLL"
    Prefix="VGA"
    Index=Dword:1

[HKEY_LOCAL_MACHINE\Drivers\PCMCIA\Detect\60]
    Dll="VOYAGER.DLL"
    Entry="DetectVGA"
```

Pocket PowerPoint application needs registry keys in order to use secondary display adapters. During initialization, the driver should update the **HKEY_LOCAL_MACHINE\Drivers\Display\Active** key with the information that is appropriate for its driver. This includes information about the native driver .dll name, as well as Buffer and Tap information. For removable display adapters, this initialization should take place when the adapter is connected to the system and the Device Manager calls the stream driver's **_Init** function. The following example shows how this is done.

```
[HKEY_LOCAL_MACHINE\Drivers\Display\Active\Voyager]
    Dll="PCARDVGA.DLL"
    BufferMode=Dword:0
    Tapmode=Dword:0
```

The following table shows the four available values for **BufferMode**.

Value	Description
bmNotShared	There is no common frame buffer.
bmTopHalf	The top half of the secondary display is also shown on the system's built-in LCD display.
bmSquashed	A scaled-down version of the secondary display is also shown on the system's built-in LCD display.
bmOff	The system's built-in display is turned off while the driver is active.

The following table shows the three available values for **Tapmode**.

Value	Description
tmNone	The display driver performs no conversion of any tap coordinates.
tmScaled	The display driver scales tap coordinates to match the secondary display's resolution.
tmUndefined	The display driver always reports tap coordinates of (0,0), but still reports tap events.

CHAPTER 7

Universal Serial Bus Drivers

The universal serial bus (USB) is an external bus architecture for connecting USB-capable peripheral devices to a host computer. USB is not designed to be used as the internal bus for connecting CPUs to main memory and to devices that reside on a motherboard. Instead, USB is a communication protocol that supports serial data transfers between a host system and USB-capable peripherals. USB technology was developed as a solution to the increasing user demands on computers and the need for flexible and easy-to-use peripherals. USB technology directly affects a number of standard peripherals, such as keyboards, joysticks, mouse devices, digital cameras, computer-telephony integration (CTI), and video-conferencing products.

USB offers the following benefits to system designers and users:

- USB provides a single, well-defined, standard connector type for all USB devices. This simplifies not only the design of USB devices, but also a user's task of determining which plugs correspond to which sockets.
- USB eliminates the need for separate mouse, modem, keyboard, and printer ports, thus reducing hardware complexity.
- USB supports hot plugging, which means that USB devices can be safely connected and disconnected while the host is turned on. Other generic peripheral connection standards, such as the Small Computer System Interface (SCSI), require that the host be turned off when peripherals are added or removed.
- USB supports Plug and Play. When a USB device is plugged in, the host computer identifies the device and configures it by loading the appropriate driver.
- USB provides flexibility in how devices are powered. USB devices can draw power directly from the USB cable (bus-powered devices), supply their own power from batteries or from a wall socket (self-powered devices), or use a combination of both types of power.
- USB supports power-saving suspend and resume modes.

- USB offers a high-speed 12-megabits-per-second (Mbps) mode and a low-speed 1.5-Mbps mode that support a variety of peripherals.
- USB guarantees certain amounts of bandwidth for devices that cannot tolerate non-continuous transmission that comes in bursts, such as streaming audio and video devices.
- USB offers four different data transfer types that are suited to the needs of various types of peripheral.
- USB enables multiple peripherals to communicate simultaneously with the host.

Consult the following sources for additional information about USB technology that is important both for OEMs who add USB support to their Windows CE–based platforms and for independent hardware vendors (IHVs) who build USB peripherals:

- USB Implementers Forum Web site

  This site contains the complete USB specification, *Universal Serial Bus Specification, Revision 1.0.*
- Intel Corporation Web site

  This site contains information on USB hardware and microcontroller chips, such as the 8x930Ax and 8x931xA series chips.

Note The official *Universal Serial Bus Specification, Revision 1*, uses the term function to refer to USB-capable peripheral devices. However, because function typically refers to callable units of C/C++ code, Windows CE documentation uses the term USB device to refer to USB peripherals. In addition, the official *Universal Serial Bus Specification, Revision 1*, uses the term USB client driver to refer to device drivers for USB devices, but to avoid confusion with client/server terminology, this documentation uses the term USB device driver.

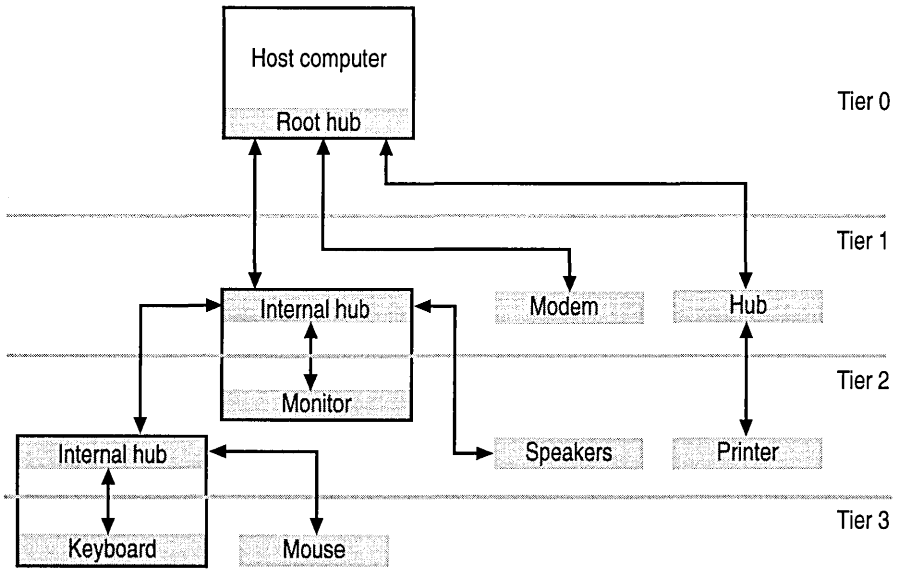
USB Architecture

A USB system consists of a host computer, one or more USB devices, and a physical bus. The host consists of two layers: an upper software layer, which includes USB device drivers, and a host controller hardware layer, also known as an adapter layer. The main responsibility of the host computer is to control data transfers to and from USB devices. USB devices are peripherals that use the USB electrical and data format specifications to communicate with the host computer. The physical bus is the set of USB cables that links the controller with the peripherals.

USB Topology

USB is a tree-structured bus, which in the vocabulary of the *Universal Serial Bus Specification, Revision 1*, is a star-tier topology. The host computer contains a single root node, or hub, of the USB tree. This hub mediates between its host computer and any peripheral devices. Hubs have exactly one connection—called an upstream port—to higher levels in the USB tree. Hubs can have up to seven downstream ports for connecting peripheral devices and other hubs. By connecting hubs together, up to 127 devices can be attached to the host computer. Peripheral devices are always leaf nodes within a USB bus. However, as a matter of practical implementation, many USB peripheral devices have hubs integrated into them, so users typically do not need to purchase separate USB hubs.

The following illustration shows a USB bus with several common peripherals connected. This illustration is modeled after the diagram of a typical USB bus configuration in the *Universal Serial Bus Specification, Revision 1*, but with the hubs and peripheral devices represented more explicitly.



The association of the mouse with the keyboard's internal hub and the speakers with the monitor's internal hub is arbitrary. For example, a user could instead connect the mouse to the monitor's internal hub, the modem to the keyboard's internal hub, and the speakers to the stand-alone hub in Tier 1 without affecting the system's functionality and without having to reconfigure any software on the host computer.

USB Transfer Types

Windows CE 2.10 supports all four types of data transfer defined in the *Universal Serial Bus Specification, Revision 1*. Device drivers for USB devices can use any of the following transfer types, as appropriate:

- Control transfers

  Control transfers are bidirectional transfers that are used by the USB system software mainly to query, configure, and issue certain generic commands to USB devices. Control transfers can contain 8, 16, 32, or 64 bytes of data, depending on the device and transfer speed. Control transfers typically take place between the host computer and the USB device's endpoint 0, but vendor-specific control transfers may use other endpoints.

- Isochronous transfers

  Isochronous transfers provide guaranteed amounts of bandwidth and latency. They are used for streaming data that is time-critical and error-tolerant or for real-time applications that require a constant data transfer rate. For example, an Internet telephony application that carries a conversation in real time is a good candidate for isochronous transfer mode. Isochronous data requires guaranteed amounts of bandwidth and guaranteed maximum transmission times. For isochronous transfers, timely data delivery is much more important than perfectly accurate or complete data transfer.

- Interrupt-driven transfers

  Interrupt-driven transfers are used mainly to poll devices to check if they have any interrupt data to transmit. The device's endpoint descriptor structure determines the rate of polling, which can range from 1 through 255 milliseconds. This type of transfer is typically used for devices that provide small amounts of data at sporadic, unpredictable times. Keyboards, joysticks, and mouse devices fall into this category.

- Bulk transfers

  Bulk transfers are for devices that have large amounts of data to transmit or receive and that require guaranteed delivery, but do not have any specific bandwidth or latency requirements. Printers and scanners fall into this category. Very slow or greatly delayed transfers can be acceptable for these types of device, as long as all of the data is delivered eventually.

USB Host Controller

The host controller, or adapter, is a hardware layer that is contained within the host computer. The host controller converts data between the format that is used by the host computer and the USB format. Only OEMs who implement Windows CE–based products that use USB need to write drivers for USB host controllers. For more information, see "Developing Native Device Drivers."

USB Devices

USB peripheral devices consist of one or more physical components that implement the abilities of the devices. These components are called *interfaces*. Each interface typically provides some useful grouping of functionality, but exactly what constitutes an interface is an implementation detail. For example, a USB mouse device could present one interface for horizontal and vertical movement information and a separate interface for left and right button information. As another option, the device could present a single interface containing all of the information. Both are valid approaches, but each approach has implications for how the device driver must operate.

Associated with each interface is a set of endpoints. Endpoints are the ultimate producers or consumers of data that is transmitted across the bus. All USB devices have a special endpoint, known as endpoint 0, which supports the generic USB status and configuration protocol.

USB device drivers establish logical communication channels, called pipes, to the various endpoints on a USB device. A pipe is a software association between a USB device driver and an endpoint. Pipes can be thought of as communication channels that use function calls to the USB system software to communicate with their associated endpoints. The characteristics of a pipe, such as the direction of communication and the required bandwidth, are determined by the endpoint characteristics, which in turn are indicated in the endpoint descriptor structure.

The bus interface hardware on a USB device is responsible for the transmission and reception of USB-structured data. The logical USB device corresponding to a physical USB device consists of USB abstraction entities, such as the device endpoints and their corresponding pipes.

USB System Software

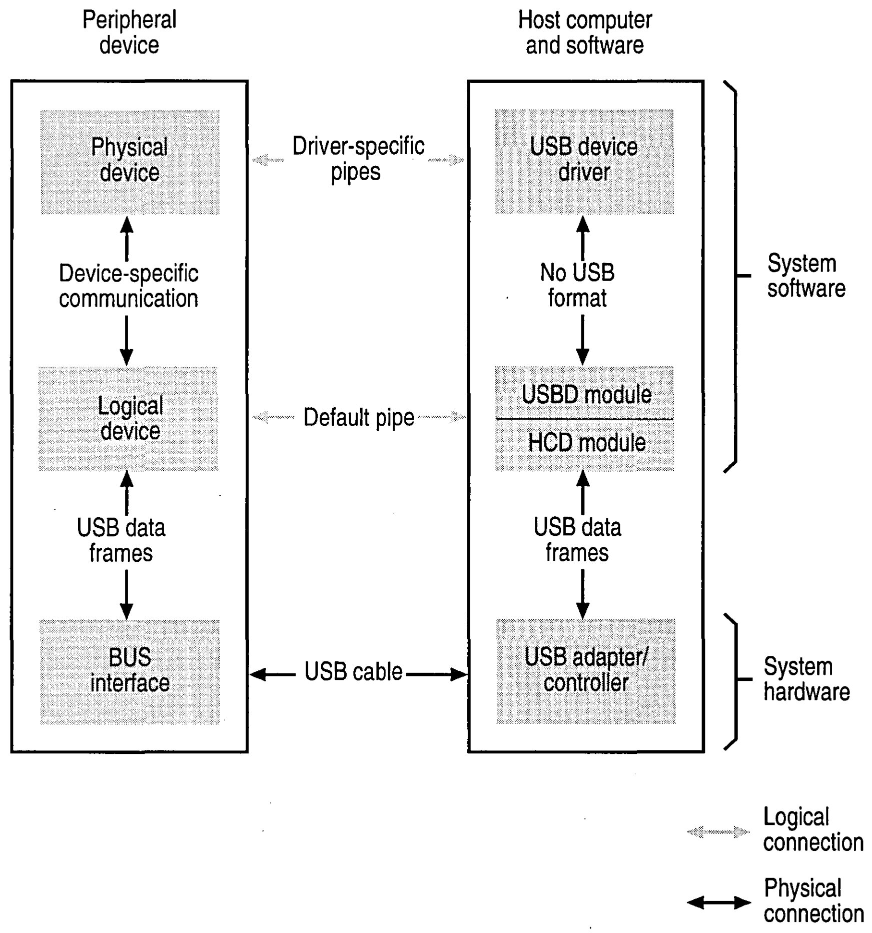
USB system software consists of two layers: an upper layer of USB device drivers and a lower layer of USB functions that are implemented by Windows CE. USB device drivers use the USB functions to establish connections to the devices they control and to configure and communicate with the devices. The lower layer of USB functions performs several interrelated tasks:

- Manage all communication between USB device drivers and the host computer's built-in USB root hub
- Load and unload USB device drivers at the appropriate times
- Translate data to and from the USB protocol's frame and packet formats
- Perform generic configuration and status-related tasks by establishing communication with the generic endpoint on all USB devices

The lower layer is itself composed of two parts—the upper universal serial bus driver (USBD) module and the lower host controller driver (HCD) module. The USBD module implements the high-level USBD interface functions in terms of the functionality provided by the HCD module. USB device drivers use the USBD interface functions to communicate with their peripherals.

IHVs and manufacturers of USB devices should make use of the functions that are provided by the USBD to implement their USB device drivers. OEMs are responsible for providing an HCD module to their Windows CE–based platforms so that their hardware properly interfaces with the USBD module.

The following illustration shows the two layers of software in the context of the host's USB hardware and a peripheral device.



During a data transfer, the flow of operation typically proceeds in the following sequence:

1. A USB device driver initiates transfers by using USBD interface functions to issue requests to the USBD module.
2. The USBD module divides requests into individual transactions, based on its knowledge of the bus and on characteristics of the USB devices that are connected to the bus.
3. The HCD module schedules these transactions over the bus.
4. The host controller hardware performs or completes the transactions.

All transactions on the bus originate from the host side; the peripherals are totally dependent.

The following sections on USB system software describe the various components of USB support in Windows CE 2.10. The primary goal of the USB support provided by Microsoft, aside from enabling IHVs to write device drivers for USB devices, is to help OEMs expand existing USB support on their platforms. Currently, USB support includes only the host side of the USB specification, which enables Windows CE to support USB peripherals. OEMs are free to add device-side support, which would enable Windows CE–based platforms to serve as USB peripherals to other USB hosts.

Supported and Unsupported USB Features

Windows CE 2.10 supports the following USB features:

- Bus enumeration

  Windows CE supports enumeration of USB devices on the bus. The bus enumeration process is a multistep query sequence: the HCD module acquires information from a connected device, assigns it a unique USB address, and sets a configuration value. Once enumeration is complete, the device is configured and ready to conduct, transmit, and receive transactions. At this point, the USBD module attempts to load one or more USBDs to control the device, based on the information contained in the device and interface descriptors. If no suitable driver has been registered for the device, a user is prompted to enter the name of a driver to control the device.

- Power management

  Windows CE provides support for bus-powered and self-powered devices. For both types of device, the USBD module reads the power requirements of the device from the descriptor information and rejects the device if it exceeds the maximum power threshold. OEMs can set the current-draw limit, so IHVs should not rely on any particular amount of available current, except as detailed in the *Universal Serial Bus Specification, Revision 1.*

- Transfer types

  Windows CE supports all four types of data transfer defined in the *Universal Serial Bus Specification, Revision 1.* USB device drivers can use any of the transfer types that are appropriate for their peripherals. However, Windows CE 2.10 does not support the control transfers defined in the *Universal Serial Bus Specification, Revision 1* to put a USB device into the suspend state. Depending on device capabilities, however, a USB device driver may be able to suspend the device by using the **SetConfiguration** function to deconfigure the device.

- Class drivers

  The USB architecture implemented in Windows CE supports loading class drivers, although Microsoft does not supply any sample class drivers. Examples of device classes include the stream class and the human interface device (HID) class, among others. OEMs or IHVs can write their own class drivers and load them appropriately, using the registry mechanism.

There is no support on Windows CE 2.10 for making a Windows CE–based platform itself appear as a USB peripheral to other host computers. That is, the HCD and USBD modules supplied in Windows CE do not provide facilities to connect a Windows CE–based platform to a desktop computer that is running as a USB host.

USB Power Management

Windows CE provides full support for power management of USB devices, as described in the *Universal Serial Bus Specification, Revision 1.* Very important for Windows CE are support for suspending and resuming, because Windows CE–based platforms have a power-on and startup cycle that differs from the one on desktop computers. Support for bus-powered and self-powered USB devices is also important because many Windows CE–based platforms have limited power resources. For more information about power management, see "Developing Native Device Drivers."

Windows CE supports power cycling USB devices in association with the standard Windows CE power states. When Windows CE issues a POWER_DOWN notification, the HCD module suspends the USB host controller hardware and all devices. When power returns to the platform, Windows CE sends a POWER_UP notification to the HCD module. When the host controller hardware has been reinitialized, the USBD module unloads the USB device drivers loaded prior to the POWER_DOWN notification, identifies all the USB devices that are currently connected to the bus—a process called bus enumeration—and loads the USB device drivers for those devices. This power cycle processing is very similar to that performed by the Windows CE Device Manager for PC Card devices.

This implies that USB device drivers may need to take special action to make a power cycle transparent to upper-level applications. For example, if a USB device provides a file system, the device driver should preserve open file handles across a power cycle. There are several ways to accomplish this. One solution is for the USB device driver to register itself with the Device Manager as a stream interface driver by calling the **RegisterDevice** function. This increments the reference count on the USB device driver's dynamic-link library (DLL) so that when the USBD module unloads the driver, the driver's code still remains in memory. The USB device driver could keep any application file handles open and wait for the the call to the **USBDeviceAttach** function, which occurs after the system resumes and the USB device is ready to be used. The disadvantage of this approach is that the driver remains in memory even after the USB device is detached from the system. The second solution is to separate the USB interface from the upper-level file system interface. For example, the PC Card file allocation table (FAT) file system module uses this approach to separate its file system driver code, which must manage file handles, from the PC Card driver code that actually manages the PC Card hardware.

Windows CE versions 2.12 and later provide full support for bus-powered and self-powered USB devices. When a user connects any self-powered or bus-powered device to a Windows CE–based platform, the USB system software automatically accepts or rejects the device, based on the device's power requirements and the system's overall power load. The power model is identical for both self-powered and bus-powered devices.

When a USB device is attached to a platform, the HCD module sets the initial power configuration. During the device attachment processing phase, the HCD module reads the power requirements of the USB device configurations from the device configuration descriptor structures. It then calls in to the platform-specific portion of the HCD module to determine if the host platform can support the USB device's power requirements. An OEM can implement code in the platform-specific portion of the HCD module to test system power status, such as whether the system is running on batteries or is plugged into a power outlet, to assist in making this determination. In this way, the HCD module can choose an appropriate power configuration for the USB device from those listed in the device's configuration descriptor structures.

At this time, Windows CE 2.10 does not support placing a USB peripheral into suspend mode programmatically.

Writing USB Device Drivers

This section describes how to write device drivers for USB devices running on Windows CE. USB device drivers exist to make the services of peripheral devices available to applications. Although there are no standard mechanisms that USB devices must use to accomplish this, there are various strategies that USBDs can adopt, depending on the nature of the peripherals that they control:

- Use the stream interface functions

  A USBD can expose the stream interface functions. Applications can then treat the peripheral device as a file and use standard file I/O functions to interact with the device. However, because the Device Manager is not involved in the loading and unloading of USBDs, any USBD that exposes the stream interface functions must register and deregister its special device file name manually, using the **RegisterDevice** and **DeregisterDevice** functions. These functions should be called when the USBD is loaded and unloaded, respectively.
- Use the existing Windows CE application programming interface (API)

  By interacting with a Windows CE API, USB device drivers can indirectly expose certain types of peripherals to applications if Windows CE has an existing API that is appropriate to the peripheral. For example, USBDs for mass storage devices, such as hard drives and CD-ROM drives, can expose such devices through the standard installable file system interface. The sample USB mouse driver also uses this strategy. The driver does not expose the mouse device directly to applications; rather, it interacts with existing Windows CE APIs to submit the correct input events to the system. Thus, the USB nature of the mouse device is transparent to applications.
- Create a custom API specific to a particular USBD

  This strategy does not place any restrictions on the way that a USBD exposes a device. It allows you to create an API for the device that best maps to the ways that applications are likely to use it. However, you must provide appropriate documentation to application writers so that their applications can use the driver.

USBD Interface Functions

USB device drivers interact with the peripherals that they control by using the USBD interface functions. These functions, which are provided by the USBD module, constitute the core of a USB device driver's functionality. There are several categories of USBD interface functions, each related to a different aspect of interacting with a USB device, including transfer, pipe, frame, and configuration functions, as well as functions for performing other miscellaneous tasks.

Transfer functions are the most important category because they handle sending data to and receiving data from a USB device. There are four basic types of transfer: control, bulk, interrupt, and isochronous. For convenience, there are several special transfer functions that provide common types of control transfer, such as device configuration and setup requests. All transfer functions have an optional callback parameter. If a callback function is provided, the transfer functions return immediately, or asynchronously, without waiting for the transfer request to complete. When the transfer request finishes, the transfer function calls the function that is pointed to by the callback parameter. Additionally, a USB device driver can use the USB_NO_WAIT flag to cause the transfer functions to return asynchronously, even if no callback is specified. USB device drivers typically do this in situations when they queue several requests to the device and wait for only the last one to finish. In such cases, the device driver is still responsible for closing all transfer handles. If the USB device driver provides no callback function and does not use the USB_NO_WAIT flag, the transfer function blocks until the transfer is complete.

The USBD module calls the callback function in a context that may block other USB operations. Therefore, these callback functions should perform very minimal processing, preferably just setting some state variables and signaling an event so that any substantial post-processing can be handled by another thread. In particular, such callback functions cannot call any USBD interface functions. See the HID and mouse sample drivers for examples of using callback functions.

Internally, the USB system is optimized for a page size of 4 KB. If a platform has a different page size, the USB system allocates an internal, contiguous 4-KB buffer to use for data transfers. However, copying data into and out of this buffer may impose an unacceptable limit on the performance of a USB device driver. In such cases, the driver may pass an optional physical memory address that the USB system uses directly for data transfers. A driver-specific buffer used in this way must be contiguous within 4-KB segments and must not be accessed by the USB device driver during transfer operations. The **LockPages** function can be used to obtain physical address information.

The following list shows the transfer functions:

AbortTransfer	**IssueControlTransfer**
CloseTransfer	**IssueInterruptTransfer**
GetIsochResults	**IssueIsochTransfer**
GetTransferStatus	**IsTransferComplete**
IssueBulkTransfer	**IssueVendorTransfer**

The following list shows the pipe functions that open and close communication channels between a USBD and a USB device:

AbortPipeTransfers

ClosePipe

IsDefaultPipeHalted

IsPipeHalted

OpenPipe

ResetDefaultPipe

ResetPipe

The following list shows the frame functions that control how the USBD module packages data into frames for transmission on the bus:

GetFrameLength

GetFrameNumber

ReleaseFrameLengthControl

SetFrameLength

TakeFrameLengthControl

The following list shows device configuration functions for specific kinds of data transfers defined in the *Universal Serial Bus Specification, Revision 1*:

ClearFeature	**SetDescriptor**
GetDescriptor	**SetFeature**
GetInterface	**SetInterface**
GetStatus	**SyncFrame**

The following list shows miscellaneous functions for tasks related to interacting with USB implementation:

FindInterface	**RegisterClientDriverId**
GetDeviceInfo	**RegisterClientSettings**
GetUSBDVersion	**RegisterNotificationRoutine**
LoadGenericInterfaceDriver	**TranslateStringDescr**
OpenClientRegistryKey	**UnRegisterNotificationRoutine**

Required Entry Points for USB Device Drivers

All USB device drivers must expose certain entry points in their DLLs to interact properly with the USBD module. The following entry points not only enable the USBD module to connect a driver with its peripheral, but also enable a driver to create and manage any registry keys that it may need:

- **USBDeviceAttach**

  The USBD module calls this function when the USB device is connected to a host computer. The driver's implementation of this function can decline to control the device, in which case the USBD module attempts to find another driver to handle the device. A driver rarely declines to control a device, though it might under certain conditions. For example, the driver might decline to control the device if the driver can determine from the device configuration block that the device is newer than the driver. By doing so, the driver indirectly gives a user an opportunity to enter the name of the correct USBD DLL if Windows CE cannot locate another driver for the device. For more information, see "USB Device Driver Attach Processing."
- **USBInstallDriver**

  This function, which is called the first time that the USB device driver is loaded, allows the driver to create any registry keys that it needs. For more information on the format of these keys and how they are used to load USBDs, see "USB Device Driver Load Process."
- **USBUnInstallDriver**

  This function is called when a user removes the driver from a Windows CE–based platform. It is responsible for removing all registry keys that are created by the driver's **USBInstallDriver** function and for releasing any other resources that are held by the driver. For more information, see "Removing USB Device Drivers."

Registry Keys for USB Device Drivers

Registry keys control how USB device drivers are loaded. When a USB device is attached, a USBD module loads the appropriate USBD to control that device, based on the device's configuration and interface descriptor information. The USBD module locates the correct driver by using a set of registry keys, which track both the drivers and the devices. The registry keys are stored as subkeys of the **HKEY_LOCAL_MACHINE\Drivers\USB\LoadClients\.** key.

This loading method provides a flexible framework that allows drivers to be loaded in different contexts, depending on the range of devices that they are able to control. For example, OEMs may decide to include a generic class driver with their Windows CE–based platform that can control a broad range of USB devices. However, an IHV may have a driver for the a specific USB device within that class that is more efficient or works better than the generic class driver.

In this case, the IHV's driver could control a subset of the devices that the generic class driver controls, while allowing other devices to continue to be controlled by the generic driver. The structure of the **LoadClients** key defines a framework in which programmers can specify driver precedence in great detail. The following are examples of the contexts that can cause specific USB device drivers to be loaded:

- To match every device that is connected to the bus. For example, a driver that displays an icon in the system's taskbar can be loaded to control each USB device that is connected to the system. In this case, the driver could purposely fail the **USBDeviceAttach** call after obtaining the device information, in which case the USBD module would continue looking for a driver to control the device. Only one driver of this type can be registered in the system.
- To match a vendor-specific identifier. This is used for USB devices that do not fall into any currently defined class, or to provide an enhanced driver for a particular IHV's USB devices. It is not recommended for general-purpose drivers that may be able to control devices from multiple IHVs.
- To control devices of a specific device class. For example, you could write a USB device driver to control all HID class devices. The USB Working Group has defined other device classes, including communications, audio, and mass storage.
- To control each interface on a device. For example, a USB CD-ROM drive that has an audio interface, as is common with CD-ROM drives that can play audio CDs, could have separate drivers for each interface. This method is the best for loading USB device drivers because it allows compound devices that require multiple drivers to operate without additional reconfiguration.

The registry key for a USBD should either be part of an OEM's platform .reg file or be created when a USB device driver is installed on a Windows CE–based platform. At installation time, the key can be created either by a setup application or by the driver's **USBInstallDriver** function. **USBInstallDriver** should create the keys indirectly, by calling the **RegisterClientSettings** function, rather than by invoking the Windows CE registry APIs. Installation by **USBInstallDriver** occurs when an unrecognized USB device is connected to the bus and the USBD module queries a user for the name of the device driver DLL. The USBD module then loads the driver and calls its **USBInstallDriver** function.

Subkeys for each driver have the form ***Group1_ID\Group2_ID\Group3_ID\DriverName***. Each of the group identifier subkeys can be named **Default** to indicate that the USBD should be loaded if the remaining group identifier subkey names match the USB device. Otherwise, the group identifier subkey names are formed from combinations of vendor, device class, and protocol information, separated by underscores. This information comes from the USB device descriptor.

The following table shows the allowable combinations.

Group key	Allowable forms
Group1_ID	*DeviceVendorID,* *DeviceVendorID_DeviceProductID,* *DeviceVendorID_DeviceProductID_DeviceReleaseNumber*
Group2_ID	*DeviceClassCode,* *DeviceClassCode_DeviceSubclassCode,* *DeviceClassCode_DeviceSubclassCode_DeviceProtocolCode*
Group3_ID	*InterfaceClassCode,* *InterfaceClassCode_InterfaceSubclassCode,* *InterfaceClassCode_InterfaceSubclassCode_InterfaceProtocolCode*

The following code example shows the registry key setup for the sample mouse driver.

```
[HKEY_LOCAL_MACHINE\Drivers\USB\LoadClients]
  [Default]
    [Default]
      [3_1_2]
         [Generic_Sample_Mouse_Driver]
           "DLL"="USBmouse.dll"
```

This code example shows that the driver contained in Usbmouse.dll called **Generic_Sample_Mouse_Driver** is loaded by default for any interface on a USB device with an *InterfaceClassCode* of 3 (HID class), *InterfaceSubclassCode* of 1 (boot interface subclass), and *InterfaceProtocolCode* of 2 (mouse protocol). These values are defined in the USB HID specification.

The following code example shows the settings for the sample HID driver.

```
[HKEY_LOCAL_MACHINE\Drivers\USB\LoadClients]
  [Default]
    [Default]
      [3]
         [Generic_Sample_Hid_Class_Driver]
           "DLL"="USBHID.dll"
```

This example shows that the driver contained in Usbhid.dll is called **Generic_Sample_Hid_Class_Driver** and is loaded for any interface with an *InterfaceClassCode* of 3.

According to the precedence rules for loading USB device drivers, if the settings for both the sample mouse and the HID drivers are included in the registry, the HID driver is loaded first because it has the more general **Group3_ID** subkey.

USB Device Driver Load Process

The USBD module takes the following steps when loading drivers, stopping as soon as it finds a driver that accepts control of the device. The following rules describe the algorithm that the USBD module uses to search for USB device drivers. In the descriptions, **Group*X*_ID** refers to a key with the specified group set to one of the forms described in "Registry Keys for USB Device Drivers" and the remaining groups set to **Default**. If multiple drivers are registered within the same group, the one that contains the simplest form is loaded first. For example, a driver specifying a **Group1_ID** with device class code only, such as **Default\*DeviceClass*\Default**, loads before a driver specifying a **Group1_ID** with device class and subclass code, such as **Default\DeviceClass_Subclass\Default**. This allows Windows CE to conserve resources by loading as few drivers as possible. This procedure takes the following steps:

1. The USBD module searches for a subkey with the name **Default\Default\Default**. If present, the module loads the driver listed within the **Default\Default\Default\*DriverName*\DLL** subkey. A driver registered in this way is loaded for all USB devices that are connected to the system.
2. The USBD module searches for a vendor-specific driver. Vendor-specific drivers are identified by searching for the most general ***Group1_ID*** subkey that matches the device descriptor information. The most general subkey is the one that has a matching ***Group1_ID*** subkey containing the simplest allowable form and **Default** for the ***Group2_ID*** and ***Group3_ID*** subkeys. If a matching subkey is found, the module loads the driver that is listed within the subkey's ***DriverName*\DLL** subkey. For more information on allowable forms, see "Registry Keys for USB Device Drivers."
3. The USBD module searches for a device class-specific driver. Class-specific drivers are identified by searching for the most general ***Group2_ID*** subkey. If a matching subkey is found, the module loads the driver listed within the subkey's ***DriverName*\DLL** subkey.

The searches in steps 1 through 3 may not yield a matching USBD to control the device as a whole; that is, the device may have multiple interfaces, but no driver identified in steps 1 through 3 may match all of the interfaces present on the device. If so, the USBD module takes the following steps to search for matching drivers for each interface present on the device, searching for the most general **Group3_ID** subkey. If the USBD module finds a matching subkey, it loads the driver listed within the subkey's ***DriverName*\DLL** subkey.

Finally, if no appropriate USBD is located, the USBD module prompts a user to enter the name of a DLL containing the correct driver. The USBD module then loads the driver and calls the driver's **USBInstallDriver** function.

USBInstallDriver should create an appropriate subkey for the driver by calling the **RegisterClientSettings** function so that the next time that the USB device is attached, the USBD module can locate the correct driver without prompting a user.

In some cases it may be necessary to specify the precedence order to a greater level of detail; for example, combining a vendor and device class specifiers. In these cases, the **Group*X*_ID** values may be combined to generate other combinations. The precedence for such combinations is as follows, in descending order:

1. **Group1_ID\Default\Default**
2. **Group1_ID\Group2_ID\Default**
3. **Default\Group2_ID\Default**
4. **Group1_ID\Group2_ID\Group3_ID**
5. **Group1_ID\Default\Group3_ID**
6. **Default\Group2_ID\Group3_ID**
7. **Default\Default\Group3_ID**

If multiple drivers are registered at a particular precedence level, the USBD module loads the one with the most general form.

USB Device Driver Installation

The **HKEY_LOCAL_MACHINE\Drivers\USB\LoadClients** key must be set up correctly so that the USBD module can load the appropriate driver for a device when a device is attached to the bus. Each installed USB device driver must have a subkey within the **LoadClients** key for the USBD module to load it.

For a USB device driver that is supplied by an OEM, the OEM should configure the **LoadClients** key in a platform's .reg file to include subkeys for the driver. For third-party peripherals, however, the driver does not have appropriate subkeys when it is first connected to the platform. In this case, the USBD module fails to locate an appropriate USB device driver when the peripheral is attached to the bus. The USBD module instead displays a dialog box prompting a user to enter the name of the appropriate USB device driver DLL. The USBD then loads the specified driver to control the peripheral and calls the driver's **USBInstallDriver** function.

The USB device driver's **USBInstallDriver** function sets up the driver's subkey correctly within the **LoadClients** key so that the USBD module can load the driver the next time that the peripheral is attached to the bus. The driver does this by creating a **USB_DRIVER_SETTINGS** structure and passing it to the **RegisterClientSettings** function. The format of the **USB_DRIVER_SETTINGS** structure parallels that of the **LoadClients** key. If all the fields in a group have **USB_NO_INFO** value, the corresponding **Group*X*_ID** is set to **Default** in the registry. For example, the following example shows the **USB_DRIVER_SETTINGS** for the HID class sample driver.

```
USB_DRIVER_SETTINGS DriverSettings;
DriverSettings->dwVendorId          = USB_NO_INFO;
DriverSettings->dwProductId         = USB_NO_INFO;
DriverSettings->dwReleaseNumber     = USB_NO_INFO;

DriverSettings->dwDeviceClass       = USB_NO_INFO;
DriverSettings->dwDeviceSubClass    = USB_NO_INFO;
DriverSettings->dwDeviceProtocol    = USB_NO_INFO;

DriverSettings->dwInterfaceClass    = USB_DEVICE_CLASS_HUMAN_INTERFACE;
DriverSettings->dwInterfaceSubClass = USB_NO_INFO;
DriverSettings->dwInterfaceProtocol = USB_NO_INFO;
```

The following example shows how calling **RegisterClientSettings** with this **USB_DRIVER_SETTINGS** structure is equivalent to having set up this registry key.

```
[HKEY_LOCAL_MACHINE\Drivers\USB\LoadClients]
  [Default]
    [Default]
      [3]
         [Generic_Sample_Hid_Class_Driver]
           "DLL"="USBHID.dll"
```

In general, the client driver performs the following operations in **USBInstallDriver**:

- Registers a unique client driver identifier string by calling the **RegisterClientDriverId** function.
- Sets up the **LoadClients** registry key correctly by calling the **RegisterClientDriverSettings** function.
- Creates any driver-specific registry keys under the registry key that is returned by the **OpenClientRegistryKey** function. This is optional because many client drivers may not have any driver-specific registry settings. The registry key for driver-specific settings is **HKEY_LOCAL_MACHINE\Drivers\USB\ClientDrivers\*Client Driver Id String*.**

This location for driver-specific keys may change in future versions of Windows CE, so USB device drivers should always use **OpenClientRegistryKey** to manipulate the settings, rather than opening the registry key directly.

After **USBInstallDriver** completes these actions, it returns control to the USBD module. The USBD module once again attempts to load a client driver for the device, using the algorithm described in "USB Device Driver Load Process." At this time, the USBD module calls the USB device driver's attach routine.

USB Device Driver Attach Processing

After the USBD module loads a USB device driver for a peripheral, it calls the driver's **USBDeviceAttach** function. In this function, the driver performs the following operations:

- Determines whether it can control the peripheral. If the driver can control the peripheral, it returns TRUE to accept control. Otherwise, the driver returns FALSE to decline control. In the latter case, the USBD module continues to search for other drivers to control the peripheral.
- Loads additional USB device drivers for other interfaces that may be present on the peripheral by calling the **LoadGenericInterfaceDriver** function. If a USB device driver accepts control of a peripheral, the USBD module stops searching for drivers to control the peripheral. The driver that accepts control must then load any other USB device drivers for other interfaces present on the peripheral.
- Registers a callback function that is called when the peripheral is disconnected from the bus. USB device drivers use the **RegisterNotificationRoutine** function to register callback functions.

Removing USB Device Drivers

During the USB device driver attach processing operation, the device driver registers a callback function. When a user detaches a peripheral from the bus, the USBD module calls the callback function within that device driver. This function is called with the USB_CLOSE_DEVICE code, which is the only device notification code defined in Windows CE 2.10.

The callback function also has the option of calling other functions in the USB device driver, such as **USBUnInstallDriver**. **USBUnInstallDriver** removes all registry keys that were created by the driver's **USBInstallDriver** function and releases any other resources that are held by the driver. In this way, a user can remove any old registry settings for a particular device when a new or updated driver for that device is available. In Windows CE 2.10, the USBD module never calls **USBUnInstallDriver**.

Sample USB Device Drivers

The Microsoft Windows CE Platform Builder contains source code for a sample driver for USB mouse devices and an HID class driver. The mouse driver uses interrupt-driven transfers. OEMs and IHVs are encouraged to use the source code for these sample drivers as the basis for other USB device drivers. In the sample, the registry keys are configured to load the drivers automatically; plugging in a mouse or keyboard loads these drivers. Because the HID class driver can control mouse devices, there is no reason to include the mouse driver DLL on platforms that support USB keyboard and mouse devices.

After the sample USB mouse driver is loaded and the USBD module calls the driver's **USBDeviceAttach** function, the driver calls an initialize function that opens a pipe to the mouse device's interrupt endpoint. It also starts a worker thread to handle interrupts. This thread enters a loop in which it submits interrupt transfers by calling the **IssueInterruptTransfer** function. After the transfer completes, the driver retrieves the mouse event data from the function's *lpvBuffer* parameter. It then creates an appropriate mouse event to submit to the Windows CE input system.

There are a number of USB device classes that Microsoft does not currently supply sample drivers for. These include the audio device, storage device, communication device, physical interface device, and power device classes.

Sample HID Class Driver

The sample USB HID class driver supports input devices such as keyboards, mouse devices, joysticks, gamepads, steering wheels, and so on. In order to present applications with a consistent method of accessing those devices, the HID class driver uses the Microsoft® DirectInput® API. The HID class driver uses the USB interrupt transfer and control transfer functions to access USB input devices. It uses the stream interface functions to interact with the DirectInput subsystem.

In addition to sending input events from HID class devices to the DirectInput subsystem, the HID class driver also always generates ordinary keyboard and mouse events by using the **keybd_event** and **mouse_event** functions. This is not necessary for interacting with the DirectInput subsystem, but it does mean that if an application that is using the mouse or keyboard fails or stops responding, the use of those devices is still available to other applications.

The DirectInput subsystem in Windows CE uses I/O control codes to request various actions from the HID class driver. The following table shows the eight I/O control codes that the HID class driver must support in its **IOControl** function.

I/O control code	Description
IOCTL_HID_ACQUIRE	Notifies the HID class driver that the DirectInput subsystem no longer wants to receive events

	connected to a handle that was previously obtained through use of the IOCTL_HID_SET_FORMAT control code.
IOCTL_HID_ATTACH	Notifies the HID class driver that an application needs to use a device.
IOCTL_HID_ENUM_DEVICES	Return a list of devices that the HID class driver is currently managing.
IOCTL_HID_ENUM_OBJECTS	Return the number of input sources, such as buttons, on the device.
IOCTL_HID_GET_INST_DATA	Returns data from the device to the DirectInput subsystem.
IOCTL_HID_POLL	Requests that the HID class driver poll the device to get the device's status.
IOCTL_HID_SET_FORMAT	Sets the format of data that the HID class driver returns for a device to the DirectInput subsystem, and acquires the device for use by an application.
IOCTL_HID_TRANSFER_EVENT	Registers an event handle that the HID class driver can use to notify the DirectInput subsystem when the device generates data.

The implementation of the DirectInput subsystem for Windows CE is a subset of the full DirectInput API that is defined for desktop versions of Windows. The following differences between these implementations affect the HID class driver:

- Windows CE supports only foreground-exclusive mode.
- Attempting to enumerate the available HID class devices fails if there are no such devices connected to the Windows CE–based platform because the USB subsystem unloads the HID class driver when there are no HID class devices to control.

Testing USB Device Drivers

There is no extensive USB test suite for Windows CE at this time. The sample USB mouse and HID drivers, the USB CD-Changer device driver for an Auto PC device, and the USB 8x930Ax peripheral kit and evaluation board from Intel Corporation can be used to assist in testing USB scenarios. These are the methods used at Microsoft to test the USB system software for Windows CE version 2.10. Further details on testing a USB system and the device drivers on an OEM platform are available in the Platform Builder.

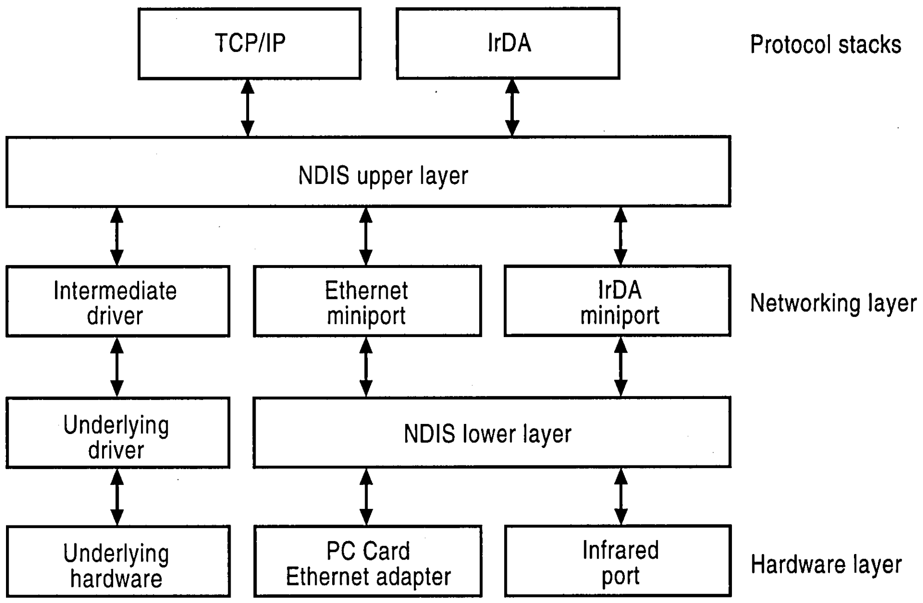
CHAPTER 8

NDIS Network Drivers

The network driver interface specification (NDIS) is the mechanism by which the Windows CE operating system (OS) supports network connectivity. NDIS provides a pair of abstraction layers that are used to connect networking drivers to protocol stacks, such as TCP/IP and Infrared Data Association (IrDA), and to network adapters, such as Ethernet cards. NDIS presents two sets of application programming interfaces (APIs) for writers of network drivers: one set interfaces to the networking protocol stacks and one set interfaces to network interface cards (NICs).

Windows CE versions 2.0 and later implement a subset of the NDIS 4.0 model that is used by Windows NT, enabling OEMs and independent hardware vendors (IHVs) to port existing Windows NT networking drivers to Windows CE. The full NDIS supports several types of network drivers, but Windows CE versions 2.0 and later support only miniport drivers, and not monolithic or full NIC drivers. In addition, Ethernet and IrDA are the only NDIS media types that are supported in Windows CE 2.10.

The following illustration shows the relationships among protocol stacks, NDIS, NICs, and miniport drivers in Windows CE.



For miniport drivers, Windows CE is largely source-code-compatible with Windows NT. This means that, with a few exceptions, Windows CE and Windows NT support identical NDIS APIs. Consult the Microsoft Windows NT Device Driver Kit for extended information on how to write a miniport driver. Because full documentation is available in the Microsoft Windows NT Device Driver Kit, this documentation does not discuss at length the process of writing miniport drivers. Miniport drivers are complex pieces of software, and for this reason Microsoft recommends that you adapt one of the sample miniport drivers or port an existing miniport driver from another OS, such as Windows NT, rather than writing one from scratch.

The Microsoft Windows CE Platform Builder includes the following sample miniport drivers:

- Proxim wireless Ethernet PC Card
- FastIR, which uses the National Semiconductor PC87338 chipset
- NE2000-compatible network adapters for PCI, ISA, and Personal Computer Memory Card International Association (PCMCIA) buses
- IrSIR infrared serial port intermediate miniport driver
- Xircom CE 2 Ethernet PC Card

For a complete list of the NDIS APIs that are supported on Windows CE, including information on the minor differences between the Windows CE API and its Windows NT counterpart, consult the Microsoft Windows CE API Reference.

NDIS Support in Windows CE

Windows CE versions 2.0 and later support the following NDIS features:

- A subset of the NDIS 4.0 API
- Ethernet (802.3) and IrDA media types
- Standard miniport drivers
- A subset of intermediate miniport drivers

  Windows CE supports intermediate drivers that use NDIS to expose a miniport interface to overlying protocol stacks and a custom interface to underlying device drivers. An example of such a driver is the IrSIR driver for infrared serial ports. Intermediate miniport drivers, unlike standard miniport drivers, do not use NDIS functions to access NIC hardware. They use NDIS functions only to access the protocol stack that they are bound to.
- Plug and Play loading of miniport drivers for PC Card–based NICs

Windows CE versions 2.0 and later do not support the following NDIS features:

- Monolithic or full NIC drivers.
- General direct memory access (DMA)

  Windows CE versions 2.0 and later do not support the NDIS functions that are related to DMA. However, developers can implement DMA code in their miniport driver for a specific combination of a Windows CE–based platform and an NIC. For more information, see “Implementing DMA for NDIS Miniport Drivers.”
- Contiguous physical memory allocations

  Miniports for built-in NICs that require contiguous physical memory for DMA transfers can have a physical block of memory reserved for this purpose in an OEM’s device memory map.
- Intermediate miniport drivers that expose both a miniport interface for use by overlying protocol stacks and a protocol interface for use by other, underlying miniport drivers
- Wide area networking through NDIS

  Protocols such as Serial Line Internet Protocol (SLIP) and Point-to-Point Protocol (PPP) are implemented by libraries that directly connect the TCP/IP protocol stack to an underlying transport medium, such as the serial port driver.

- PC Card attribute space

  Miniport drivers for PC Card–based NICs should use the PC Card Services library to access attributes and other tuples on the PC Card.
- Multipacket sends

  Multipacket sends are not supported in Windows CE versions 2.12 and earlier.

Compiling a Miniport Driver for Windows CE

The source code for miniport drivers on Windows NT is largely compatible with Windows CE. As long as a miniport driver uses NDIS functions that are supported by Windows CE, the process of porting a miniport driver to Windows CE is straightforward.

The most significant difference between Windows CE and Windows NT is that Windows CE does not support .sys or .inf files. This reduces the complexity and size of the Windows CE loader. Therefore, a miniport driver for Windows CE is compiled as a dynamic-link library (DLL) that exports the **DriverEntry** function. **DriverEntry** typically performs any general or platform-specific initializations. It also registers the miniport driver with the NDIS system by calling the **NdisMRegisterMiniport** function. **DriverEntry** must have the following prototype:

NTSTATUS DriverEntry(IN PDRIVER_OBJECT *pDriverObject*,
IN PUNICODE_STRING *pRegistryPath*);

For sample **DriverEntry** implementations, see the source code for the sample miniport drivers.

Because Windows CE does not support the Common Network .inf file format for installing a device driver, you must ensure that the proper registry keys are created for the miniport driver, typically through a setup application or through the driver's **Install_Driver** function. For more information, see "Registry Keys for Miniport Drivers."

As well as general issues that are related to the structure of miniport drivers themselves, there are also issues relating to pointers. I/O port addresses for miniport drivers are 32-bit virtual addresses that are mapped to the miniport driver's process address space. Do not cast I/O port addresses to non-32-bit types, such as **USHORT**.

A miniport driver should be installed in the \Windows directory on a target Windows CE–based platform. Miniport drivers need certain system DLLs in order to load and function correctly. Ethernet miniport drivers require the Ndis.dll, Arp.dll, and Dhcp.dll files. IrDA miniport drivers require the Ndis.dll and Irdastk.dll files. OEMs can omit some of these files if their platforms do not have the hardware that is used by a particular type of driver. For example, platforms without a built-in infrared port do not require Irdastk.dll.

Miniport drivers for PC Card–based NICs can be loaded and unloaded dynamically when the NICs are inserted or removed. This is because PC Cards are detected automatically by the Plug and Play support in Windows CE. However, other types of NIC hardware that cannot be detected automatically by Windows CE require that a platform be restarted after the miniport driver is installed in order for Windows CE to use the driver. After a platform is restarted, the NDIS component loads all drivers that have an **HKEY_LOCAL_MACHINE\Comm\*Miniport*\Group** value of NDIS listed in the registry.

Implementing DMA for NDIS Miniport Drivers

DMA is important for efficient networking because it enables the layers in the networking architecture to share data without first copying that data. Currently, Windows CE does not have any inherent DMA mechanisms; however, developers can implement equivalent functionality in their miniport drivers.

There are two categories of DMA: slave DMA and busmaster DMA. Slave DMA is appropriate for OEMs because it requires a block of memory that is pre-allocated in the device memory map of the Windows CE–based platform. Miniport drivers can map that block of physical memory to the driver's virtual memory space, and then use the **VirtualAlloc** and **VirtualCopy** functions to move data in and out of that space. For an example of this type of DMA implementation for a Windows CE OS for a Windows-based hardware development target platform, see the National Semiconductor IrDA sample driver.

Currently, no sample miniport driver implements busmaster DMA, which is a slightly more complex mechanism than slave DMA. To receive data, the driver allocates a shared memory block, transfers the data to buffers within that block, and uses NDIS functions to indicate that a packet has arrived. To send data, the driver uses the **LockPages** and **UnlockPages** functions to map the virtual memory to device memory, informs the NIC of the addresses of the data, and instructs the NIC to send the data. This method can be faster for larger data block sizes.

NDIS Protocol Binding

When a miniport driver is loaded, it must bind to an appropriate protocol stack. The binding process connects the driver to the protocol stack so that they can operate together. Protocol binding takes place through the **NdisOpenAdapter** and **NdisCloseAdapter** functions.

The TCP/IP protocol stack supports miniports for both built-in and PC Card–based Ethernet hardware. The TCP/IP protocol stack in Windows CE can bind to multiple miniport driver instances, enabling it to use multiple NICs. In Windows CE 2.10, the IrDA stack supports only a single built-in infrared port, which means that the IrDA stack binds to only a single miniport driver instance.

For PC Card–based NICs and miniport drivers, protocol binding or unbinding occurs when the NIC is inserted or removed. When the NIC is inserted into a PC Card socket, the Device Manager identifies the card, loads its driver, and binds it to a protocol stack after the driver initializes itself. When the NIC is removed from the PC Card socket, the Device Manager unbinds the miniport driver from the protocol stack, shuts down the driver, and unloads the driver's DLL.

The particular miniport instances that protocol stacks bind to are stored in the **HKEY_LOCAL_MACHINE\Comm\*Protocol*\Linkage\** registry key. For more information, see "Registry Keys for Miniport Drivers."

NDIS Power Management

Power management for PC Card–based NICs is identical to power management for other PC Cards. The miniport driver performs the same power-cycle processing that is required of all PC Card device drivers; no additional processing is necessary. After power returns, the Device Manager calls the appropriate unbind and bind functions in the miniport driver.

For a miniport driver for a built-in NIC, the miniport driver's reset function is called when power returns. Miniport drivers for IrDA must support the OID_IRDA_REACQUIRE_HW_RESOURCES and OID_IRDA_RELEASE_HW_RESOURCES messages in order to perform power management properly. A miniport for IrDA starts without any resources; when the first IrDA socket is created, the IrDA protocol stack acquires the miniport's resources. When all IrDA sockets are closed, the protocol stack releases the resources. This ensures that the miniport driver and its NIC are not consuming power when the IrDA stack is not in use. For an example of how these object identifiers are implemented, see the FastIR sample miniport driver.

In some cases, miniport drivers may need to prevent a Windows CE–based platform from suspending—for example, due to lack of user input—while the driver is engaged in active operations. In such cases, the miniport driver can use the **SystemIdleTimerReset** function to prevent Windows CE from entering suspend mode. However, if the miniport driver is active because of a high-level protocol connection, such as an open TCP/IP socket, the miniport driver generally can let the upper-level protocol stacks prevent Windows CE from shutting down. The upper-level protocol stacks ensure that the system does not shut down while there are open sockets because such connections do not survive a power cycle.

Registry Keys for Miniport Drivers

When a miniport driver is installed on a Windows CE–based platform, the setup application should create several registry keys to expose the miniport driver to Windows CE properly. Windows CE loads NDIS drivers listed within the **HKEY_LOCAL_MACHINE\Comm\** key. Subkeys within this key are named for the corresponding miniport driver.

The following table shows the subkeys that are contained in each ***Miniport*\** key.

Name	Type	Description
DisplayName	SZ	A user-friendly name for the driver
Group	SZ	The literal string "NDIS"
ImagePath	SZ	The name of the DLL containing the miniport driver
Linkage\Route	SZ	A set of ***Miniport Instance*** keys, separated by commas

The value of the **Linkage\Route** key lists additional subkeys of the **Comm\** key for each miniport instance. These subkeys in turn contain a set of keys that describes the parameters of that miniport instance. The following table shows the subkeys that are contained in each ***Miniport Instance*** key.

Name	Type	Description
DisplayName	SZ	A user-friendly description of the miniport instance
Group	SZ	The literal value "NDIS"
ImagePath	SZ	The name of the miniport driver's DLL
Parms	subkey	Subkeys for the miniport driver's parameters

The following table shows the subkeys that are contained in each ***Miniport Instance*\PARMS** key.

Name	Type	Description
BusNumber	DWORD	The bus number for the miniport instance
BusType	DWORD	The bus type of the miniport instance

BusNumber values range from 0 to one less than the number of buses on a Windows CE–based platform. Valid **BusType** values are declared in the **_INTERFACE_TYPE** enumeration in the Ceddk.h header file. Use the ***Miniport Instance*\Parms** key to store any miniport-specific values because this is the registry location that is accessed when miniport drivers call the **NdisOpenConfiguration** and **NdisReadConfiguration** functions.

The following example shows a set of registry keys for a miniport driver.

```
[HKEY_LOCAL_MACHINE\Comm]
  [NE2000]
    DisplayName="NE2000 Compatible Ethernet Driver"
    Group ="NDIS"
    ImagePath="NE2000.DLL"
    [Linkage]
      Route="NE20001, NE20002"
  [NE20001]
    DisplayName="NE2000 Compatible Ethernet Driver"
    Group="NDIS"
    ImagePath="NE2000.dll"
    [Parms]
      BusNumber=0
      BusType=8
      CardType=1
      InterruptNumber=03
      IOBaseAddress=0300
      Transceiver=3
```

PC Card miniport drivers also require an **HKEY_LOCAL_MACHINE\Drivers\PCMCIA\*Plug-and-Play ID*\** key. This key enables the driver to load correctly when its NIC is inserted into a PC Card socket. The key typically is named for the Plug and Play identifier of the PC Card, but this is not a requirement. However, if the key is not named for the NIC's Plug and Play identifier, the driver needs additional registry keys to enable the driver-detection algorithm of the Device Manager to load the driver.

The following table shows the subkeys that are contained in the **HKEY_LOCAL_MACHINE\Drivers\PCMCIA\*Plug-and-Play ID*\** key.

Name	Type	Description
DLL	SZ	The literal string "Ndis.dll"
Prefix	SZ	The literal string "NDS"
Miniport	SZ	The name of the miniport driver for the PC Card, which corresponds to the name of the registry key within **HKEY_LOCAL_MACHINE\Comm\** for the miniport driver

The following example shows the additional keys that a PC Card network adapter could have.

```
[HKEY_LOCAL_MACHINE\Drivers\PCMCIA\NICs-R-Us Inc.-Super2000-E6FE]
  DLL="NDIS.DLL"
  Prefix= "NDS"
  Miniport="NE2000"
```

Ndis.dll sets values for the **BusNumber** and **BusType** keys for PC Card–based NICs. The **BusNumber** key contains the socket and function pair for the network adapter. The **BusType** key contains the value for the PC Card bus. Finally, if the card information structure (CIS) of the PC Card contains a network address value, Ndis.dll creates a **\HKEY_LOCAL_MACHINE\Comm\*Miniport Instance*\Parms\NetworkAddress** key to store the network address.

Protocol binding for miniport drivers for PC Card–based NICs occurs automatically when the driver is loaded. Protocol binding for miniport drivers for built-in NICs is controlled by the registry keys stored within **HKEY_LOCAL_MACHINE\Comm\IrDA\Linkage\** and **HKEY_LOCAL_MACHINE\Comm\Tcpip\Linkage\**. Each of these **Linkage\** keys should contain a single key called **Bind**. You can set the **Bind** key to a list of miniport instances that the corresponding protocol stack binds to. The following example shows the values that the registry could contain to enable the IrDA protocol stack to bind to the IrSIR miniport and the TCP/IP stack to bind to both the PPP and NE2000 miniports.

```
[HKEY_LOCAL_MACHINE\Comm]
  [IrDA\Linkage]
    Bind=multi_sz:"IrSir1"
  [Tcpip\Linkage]
    Bind=multi_sz: "PPP","NE20001"
```

Testing NDIS Miniport Drivers

The Microsoft Windows CE Device Driver Test Kit for Windows CE versions 2.1 and 2.11 includes a tool called Ndistest to assist in testing NDIS Ethernet miniport drivers. The Test Kit does not include any tools for testing IrDA miniport drivers. The Ndistest tool is a special protocol driver that provides functionality and stress testing. Ndistest for Windows CE is based on the test tool for Windows NT, with the difference that the Windows CE test tool does not include a graphical user interface (GUI) to configure and run the tests. Rather, Ndistest for Windows CE requires that a user manually edit configuration files and start the tests by using a command line. Dual-computer testing for Windows CE runs against an Ndistest server running Windows NT.

Although the Test Kit is part of the Platform Builder, it was not present for Windows CE version 2.10. The Test Kit for Windows CE version 2.10 is available on the Windows CE Web site at http://www.microsoft.com/windowsce/downloads/embedded/default.asp. The Ndistest tool from the Test Kit for Windows CE version 2.1 works on both Windows CE versions 2.1 and 2.11.

CHAPTER 9

Block Device Drivers

Block device drivers are for devices that allow data to be read or written only in blocks of a fixed size. Block devices do not allow individual bytes of data to be read or written. Block sizes tend to be one or a few kilobytes; some common sizes are 512 bytes, 1 KB, 2 KB, and 4 KB. Block devices are ideally suited to mass storage and persistent storage applications, such as disk and tape drives or non-volatile RAM disks.

The most common block devices that are used with Windows CE are those that use linear flash memory chips to implement persistent storage. The material in this section, however, applies to drivers for any type of block device. Some common types of block devices are hard disks and ATA-style flash RAM disks in miniature card, PC Card, and compact flash card form factors. Block devices that conform to the industry standard Advanced Technology Attachment specification work with the Microsoft ATADisk driver, which is included with Windows CE.

Block device drivers for Windows CE are typically implemented by using the stream interface driver model. These drivers are managed by the Device Manager, and expose file I/O functions in order to interact with applications. The most important of those functions for block device drivers is the driver's **DSK_IOControl** function, which handles all I/O requests to block devices. The specific I/O control codes for **DSK_IOControl** are the same ones that Windows CE uses to interact with the file allocation table (FAT) file system driver.

Block device drivers are not required to use the stream interface driver model. They can use the file system driver model if the device that they control is to be used for file storage. OEMs or independent hardware vendors (IHVs) may also use the file system driver model to implement other functionality, such as secure encrypted file systems or extended file system name spaces.

Block Device Drivers for Linear Flash Memory Devices

The first version of Windows CE was used solely on platforms with battery-backed RAM storage with a simple file system to manage the storage. However, Windows CE versions 2.10 and later are often used on embedded systems that require persistent storage and cannot depend on battery-backed RAM storage. As a solution, flash memory has been implemented because it is the industry standard for nonvolatile storage in embedded applications.

The two leading types of flash memory architecture are ATA flash and linear flash memory. Both conform to the industry standard PC Card form factor, and they can be used interchangeably in existing PC Card sockets. However, there are significant differences between the two architectures that affect their performance on Windows CE.

ATA cards emulate the behavior of an ATA-style hard drive by means of linear flash memory components and a special microcontroller chip, which performs hardware emulation of an ATA-style hard disk. An ATA card appears as an ordinary hard drive to an operating system (OS). ATA-style hard disks are block devices, and thus ATA cards require block device drivers in order to work with Windows CE.

Linear flash memory takes its name from the fact that, unlike ATA-style flash, the individual storage locations form a contiguous range of memory addresses, each of which can be accessed directly. Thus, linear flash memory can be read directly as though it were RAM or ROM. However, linear flash memory can only be written to in blocks. Moreover, linear flash memory on Windows CE uses a software driver layer to emulate a disk drive; this eliminates the need for special controller hardware. Linear flash memory devices that operate in this block-oriented fashion use the driver layer to translate data to and from a block format that Windows CE can understand.

This driver layer is called the flash translation layer (FTL). The underlying data format used by FTL software has been adopted by the PC Card industry as an official data format. The FTL software component implemented on Windows CE is the TrueFFS driver from M-Systems, Inc. The TrueFFS driver, a stream interface driver, exposes standard Windows CE stream interface functions to the OS. Currently, the TrueFFS driver can access only a DiskOnChip built-in device from M-System, Inc. Windows CE supports linear flash memory in several form factors: minicards, industry standard PC Cards, and built-in DiskOnChip devices.

Block Device Functionality

Block devices can be closely integrated with Windows CE, providing the following functionality:

- Extension of the size of the Windows CE object store

  Block devices can extend the Windows CE object store beyond the size of the system's physical RAM. Block devices appear as folders within Windows CE Explorer. Users can perform many ordinary operations on block devices, such as drag-and-drop operations. Users can also create and delete files and directories, retrieve file and directory information, and even reformat the devices. These operations are all transparent to a user, who does not need to know that the underlying device is not the same as the system's physical RAM. Because block device drivers interface seamlessly with Windows CE, it is the OS's file system code that performs these operations on the block device for a user.

- Storing both code and data

  Users can store both application code and data files on a block device. In Windows CE 2.10 and later, the OS can use demand paging to load applications into memory, as needed. Versions of Windows CE prior to 2.10 did not load applications from external block devices, using this demand paging mechanism; instead, they loaded the entire application into RAM when it was launched.

- Interoperability between OSs

  Data on block devices such as linear flash memory cards transfers seamlessly between Windows CE, Windows 95, Windows NT, and MS-DOS. All these OSs provide support for these block devices, using their own driver layers.

- Registry storage

  Windows CE versions 2.10 and later can store registry data on block devices, although it cannot be accessed directly from those devices. Windows CE 2.10 introduced application programming interfaces (APIs) for saving and loading registry information from block devices. Registry data must first be copied to the RAM-based registry before it can be used. For more information, see "Persistent Registry Storage on Block Devices."

- Database storage

  Databases on Windows CE, such as Contacts or Tasks, can be stored and used in place on block devices. Windows CE 2.10 supports new APIs for mounting database volumes from locations other than RAM and ROM. For more information, see "Persistent Database Storage on Block Devices."

Limitations of Linear Flash Memory for Block Devices

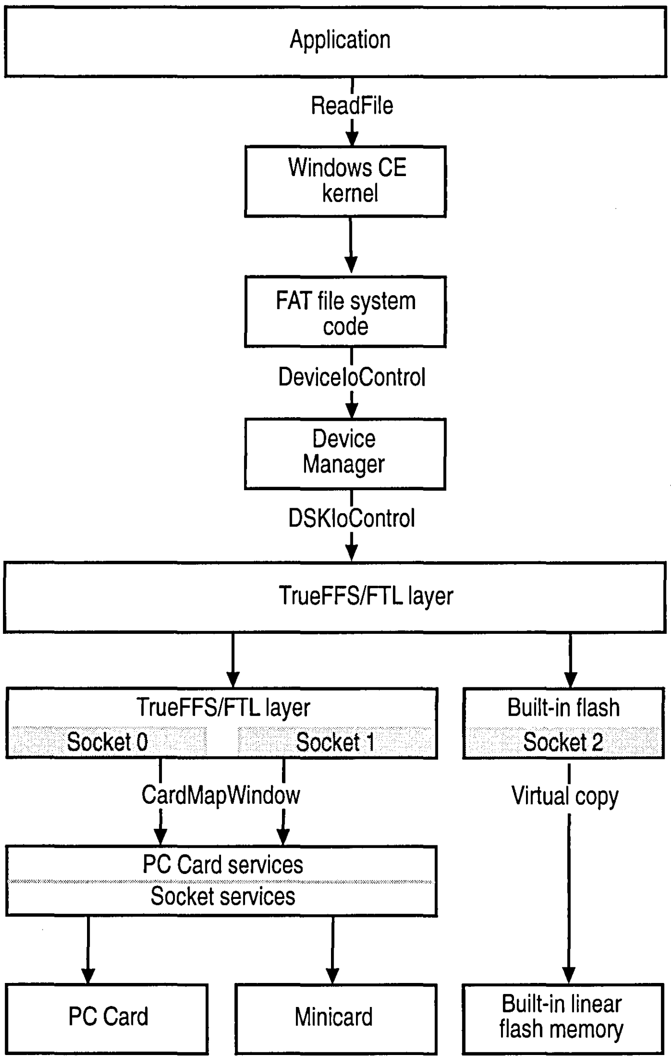
Although linear flash memory technology is very commonly used to implement block devices, there are some limitations to the hardware:

- Linear flash memory typically provides read-access times similar to ordinary DRAM but has slow write-access times and limited write-cycle lifetimes. Linear flash memory is therefore unsuitable as a replacement for true DRAM. Instead, linear flash memory is best suited for supplementary storage space, as a solid-state replacement for disk drives.
- On Windows CE, new memory technology drivers (MTDs) cannot be added dynamically to support additional types of linear flash memory hardware. Only OEMs can add support for new types of linear flash memory hardware.
- Windows CE does not inherently support resident flash arrays (RFAs) because there is no standard approach for handling tasks such as error correction, interleaving, and wear leveling. OEMs can add linear flash memory hardware directly to platforms, but they should investigate the feasibility of using a socketed form of linear flash memory, such as DiskOnChip.
- The TrueFFS driver must be a stream interface driver.

System Architecture for Block Devices

Device drivers for block devices are generally stream interface drivers, and the block devices appear as ordinary disk drives. Applications access files on a block device through standard file APIs—for example, **CreateFile** and **ReadFile**—called on the appropriate device file name, such as DSK1: or DSK2:.

The following illustration shows the flow of control from an application call to **ReadFile** to the actual read from various kinds of block devices, using linear flash memory hardware.



The application calls **ReadFile**, using a handle to a file that is stored in linear flash memory. The FAT file system, which is included with Windows CE, translates the read request to logical blocks. The FAT file system searches the buffer cache for the requested blocks. If these are not present, it issues an **IOControl** request to read bytes from the block device. The TrueFFS driver receives the **IOControl** request, and then fulfills the request by accessing the linear flash memory block device through one of the socket layers.

Block Device File Systems

Windows CE implements a separate file system—the FAT file system—to support block devices. The FAT file system does not read or write to block devices directly; it uses underlying block device drivers for all access to block device hardware. The block device driver must present the block device to the FAT file system as a block-oriented device.

Block device drivers transparently emulate ordinary disk drives, so applications do not need to behave differently when reading files from and writing files to linear flash memory. The FAT file system implements a logical file system and provides an abstraction between files in the application name space, such as \PC Card\Excel Docs\Expense report.pxl, and devices in the device name space, such as DSK1:. The block device driver is responsible for guaranteeing safe I/O operations even when interrupted by a power cycle. The FAT file system accesses the block device by calling the block device driver's **IOControl** function with the appropriate I/O control codes.

OEMs can implement additional installable file systems. Any such file systems should interact with a block device driver in the same way as the FAT file system.

Loading and Unloading a File System

Block device drivers need to perform certain tasks to work correctly with Windows CE file systems. When the driver receives DISK_IOCTL_INITIALIZED in its **DSK_IOControl** function, it also receives a pointer to a **POST_INIT_BUF** structure. This structure lists the driver's registry key, which the driver can use to retrieve the name of a file system driver. The driver must pass that name and a device-specific handle also present in the structure to the **LoadFSD** function. The Device Manager then loads the correct file system driver. The file system driver name should never be hard-coded into a block device driver; it should always be retrieved from the registry. Hard-coding the file system driver name means that your block device driver could stop working if later versions of Windows CE use different names for their file system drivers.

The block device driver should be prepared to accept I/O control requests as soon as it calls **LoadFSD**, even if **LoadFSD** has not yet returned. Because of the way the Device Manager implements **LoadFSD**, it is possible for the file system driver to be loaded and to issue a request to your block device driver before **LoadFSD** actually returns. It is also possible for **LoadFSD** to return successfully, even if there are subsequent errors initializing the file system driver. Presently, there are no mechanisms that the block device driver can use to detect such errors.

Windows CE versions 2.12 and earlier do not support multiple file system drivers acting on the same block device. If a Windows CE–based platform has multiple file system drivers installed, the block device driver must determine which one to load during its processing of DISK_IOCTL_INITIALIZED. A block device driver can make only one successful call to **LoadFSD**; calling **LoadFSD** more than once causes unpredictable behavior.

The Device Manager unloads file system drivers at the time that a block device's driver is deregistered by the **DeregisterDevice** function. Once the file system driver is loaded, the block device driver cannot force it to unload or unmount the block device from the system's directory structure.

Because file systems are frequently unloaded just prior to system shutdown, block device drivers must ensure that all write operations to the block device are fully complete before the driver's **DSK_IOControl** function returns. Failure to do so can cause corruption of files or directories.

Implementing a Block Device Driver

The following sections describe how to implement a block device driver for Windows CE. Block device drivers must export the stream interface functions, perform the correct startup sequence, support device detection, use the correct registry keys, properly respond to power cycles, and provide an **Install_Driver** function.

Block Device Driver Functions

The block device driver exposes standard stream interface functions that are common to all drivers controlled by the Device Manager. For information about these standard stream interface functions, see "Developing Stream Interface Device Drivers." In addition to the common stream interface API, the block device driver must also expose the following more specialized functions:

- **MyDriverEntry**

  An entry point function for the device driver dynamic-link library (DLL). After the Device Manager calls **LoadLibrary** to map the DLL, the system calls **MyDriverEntry**. This function performs any initialization tasks that are necessary for the block device driver. Upon successful return from **MyDriverEntry**, the Device Manager associates the driver's stream interface functions with a special device file name so that applications can access the device. The first flash memory device is named DSK1:, the second DSK2:, and so on. DSK_ is the device file name prefix for all block device drivers.

- **MyDriverCallback**

  The status callback function for notification of events from removable flash media cards. The block device driver registers this callback function with the PC Card Services driver.

- **MyDriverDetectdisk**

  The entry point for the block device driver's detection routine. **MyDriverDetectdisk** is a detection function for sockets that support hot insertion of block media. After the Device Manager loads the block device driver DLL, it calls this function to perform detection on the media. Detection typically takes place at cold start and warm start. However, for sockets that support hot insertion, detection occurs when a device is inserted into the socket.

Loading Block Device Drivers

Block device drivers are generally stream interface drivers, and are therefore loaded in the same way as stream interface drivers. OEMs who include block device drivers with their Windows CE–based platforms include keys in the platform's **\Drivers\Builtin\** registry key so that the Device Manager loads the block device driver when the platform starts. Third-party block device drivers for PC Card block devices or similar removable hardware uses the **\Drivers\PCMCIA\** registry key so that the Device Manager loads the driver when the relevant block device is connected to the system.

Registry Keys for Block Device Drivers

The Device Manager uses and manages several registry keys under the **HKEY_LOCAL_MACHINE\Drivers\** key to load, track, and unload linear block device drivers. Built-in devices, such as the DiskOnChip, rely on the registry settings to be present at startup time; therefore, these keys must be part of the default registry. However, you can install keys for removable block devices, such as linear flash memory PC Cards, upon first use of the device.

Registry Keys for PC Cards and Minicards

To detect removable block devices, the Device Manager initiates a detection sequence to determine the specific device type. A driver named MyDriver would use the registry entries shown in the following example in Windows CE version 2.1 and later. The Device Manager requires these keys to initiate the detection sequence.

```
HKEY_LOCAL_MACHINE
    [Drivers]
        [PCMCIA]
            [MyDriver]
                SZ: Prefix = DSK
                SZ: Dll = MyDriver.DLL
                SZ: IOCTL = (DWORD)4
                SZ: FSD = FATFS.DLL
                SZ: Folder = My Folder
```

```
[Detect]
    [20]
        SZ: Dll = MyDriver.DLL
        SZ: Entry = MyDriverDetectdisk
```

The value of the **Folder** key within the **MyDriver** key is optional. Its value determines the folder names that are associated with linear flash memory devices. The default name is Storage Card for the first device, Storage Card2 for the second, and so on. For example, if you use the value of the **Folder** key to change the folder name to My Folder, that is the name of the first device. The second folder is My Folder2,and so on.

The subkey **20** within the **Detect** key is provided only as an example; it can be any other number. Its value determines the order in which the Device Manager tries the driver's **MyDriverDetectdisk** function relative to other drivers' detection functions when attempting to identify an unknown type of PC Card.

Registry Keys for Built-in Block Devices

Built-in block devices, ones that are an integral part of a Windows CE–based platform, are recognized based on the **HKEY_LOCAL_MACHINE\Drivers\BuiltIn** key. A driver called MyDriver should use the registry entries shown in the following example in Windows CE version 2.1 and later.

```
HKEY_LOCAL_MACHINE
    [Drivers]
        [BuiltIn]
            [MyDriver]
                SZ: Prefix = DSK
                SZ: Dll = MyDriver.DLL
                SZ: Index = DWORD:1
                SZ: Order = DWORD:1
                SZ: IOCTL = DWORD:4
                SZ: FSD = FATFS.DLL
                SZ: WindowBase = DWORD:D0000
                SZ: Folder = My Folder
```

If you are building a Windows CE OS for a desktop Windows-based hardware development target platform, set the CEPC_DISKONCHIP environment variable to 1 prior to creation of the OS image. Otherwise, add the registry entries to one of your registry (.reg) files.

The value of the **Folder** key is optional. The value of the **WindowBase** key is provided only as an example; it can be any other 32-bit value, as specified in the documentation for your block device hardware. This value determines the location of the block device's memory window.

Installing a Block Device Driver

If a user connects an unrecognized block device to a Windows CE–based platform, the Device Manager queries a user to locate the appropriate block device driver DLL. The Device Manager then calls that driver's **Install_Driver** entry point. The **Install_Driver** function should ensure that any necessary data files are properly installed and create the relevant registry keys for the block device within the **\Drivers** portion of the registry. Block device drivers that are installed through other means, such as by an OEM as part of the software built into a Windows CE–based platform or remotely by the desktop Windows CE Services software, do not need to have an **Install_Driver** function.

Detecting a Block Device

At startup time or when a block device is connected to a Windows CE–based platform, the Device Manager calls the driver's detection function, which is listed among the driver's registry keys. The detection function attempts to recognize the type of block device hardware that is present by querying the device. If the driver recognizes the device, the Device Manager loads the block device driver into memory. This detection and load process is standard procedure, not only for block device drivers, but also for all stream interface drivers.

Accessing a Block Device

There are two methods for accessing block devices, depending on whether the device is built into a Windows CE–based platform or whether the device is removable by a user.

Drivers for built-in block devices access their devices by mapping the device's address space directly into the address space of the OS. The driver should use the **MMapIOSpace** function to map the block device into system memory.

Block device drivers for removable block devices need to use a memory window to access their devices. The driver should use the **VirtualAlloc** and **VirtualCopy** functions to create the window, which the PC Card Services library can then use to read and write data between the driver and the block device.

I/O Control Codes for Block Device Drivers

Block device drivers must respond to the I/O control codes shown in the following table to interface properly with the FAT file system. For complete information on these codes, see the Microsoft Windows CE API Reference.

I/O control code	Description
DISK_IOCTL_GETINFO	Retrieves information about the block device
DISK_IOCTL_READ	Reads data from the block device
DISK_IOCTL_WRITE	Writes data to the block device
DISK_IOCTL_SETINFO	Sets information about the block device
DISK_IOCTL_FORMAT_MEDIA	Formats at a low level or reformats the block device
DISK_IOCTL_GETNAME	Retrieves the name that file system drivers should use as the block device's folder

Power-Cycle Processing by Block Device Drivers

The Windows CE power management protocol supports power cycles that are transparent to applications. Therefore, it is important that block device drivers handle the POWER_DOWN and POWER_UP system messages efficiently.

Like all device drivers, block device drivers must limit themselves to minimal, very fast processing of the POWER_DOWN message. To accomplish this, they should save any volatile state information in RAM, set a flag to indicate that power is about to be turned off, and exit. POWER_ON processing is exactly like the processing for a regular card removal that is followed by an insertion. When power resumes, the PC Card Socket driver issues a card-removal notification for all sockets with inserted cards. Next, it checks the socket status and issues card-insertion notices for each socket with a card. Finally, the Device Manager launches its detection sequence and loads the appropriate driver for each card.

Block device drivers must detect whether they are being loaded either in response to a POWER_ON message or because a block device was disconnected from, and then reconnected to the system. This is important because the driver should preserve state information for any handles that are held by applications to files that are stored in the linear flash memory device across simple power cycles. However, if the block device is removed and replaced with a different card, any open file handles should be closed.

The following table shows how open file handles should be treated during POWER_UP processing.

Transition	Preserve open file handles
Simple power cycle	Yes
Block device removed and subsequently re-inserted	Yes, if it is possible to detect that the contents of the device have not changed
Block device removed and replaced with a different device	No

Sample Block Device Drivers

The following sections describe the programming details of two sample block device drivers in Windows CE.

Sample ATADISK Driver

In addition to the standard stream device driver functions, the ATADISK driver also exports a PC Card Plug and Play detection function, **DetectATADisk**. The detection function only reads the attribute space of the PC Card; no other data is read, and no write operations are performed on the PC Card. The function looks for disk device type 4 in the PC Card's **CISTPL_FUNCID** tuple and for ATA device type 1 in the type 1 **CISTPL_FUNCE** tuple. The presence of the **\Drivers\PCMCIA\ATADisk\** key causes the Device Manager to call the driver's **DetectATADisk** function when a PC Card is inserted and there is no driver that is associated with the card's Plug and Play identifier. The following example shows how this takes place.

```
HKEY_LOCAL_MACHINE
    [Drivers]
        [PCMCIA]
            [Detect]
                [50]
                    SZ: Dll = ATADisk.DLL
                    SZ: Entry = DetectATADisk
```

When **DetectATADisk** detects an ATA-compatible PC Card, it causes the Device Manager to load the driver listed in the **\Drivers\PCMCIA\AtaDisk\** key. The following example shows how this is done.

```
HKEY_LOCAL_MACHINE
    [Drivers]
        [PCMCIA]
            [ATADISK]
                SZ: Prefix = DSK
                SZ: Dll = AtaDisk.DLL
                SZ: IOCTL = (DWORD)4
                SZ: FSD = FATFS.DLL
```

The four values shown are required.

The following values within the **ATADISK** key are optional. If present, these values affect all devices that the ATADISK driver operates on. Some of the following values are supported so that the ATADISK driver can work with some older ATA disk devices.

Folder
: "Storage Card" causes ATADISK to report a default volume name of "Storage Card" in response to a DISK_IOCTL_GETNAME **DeviceIOControl**. You can use a different name than "Storage Card", if desired.

Cylinders
: A value of *xxx* causes the ATADISK driver to not rely on the number of cylinders that is reported by the ATA device in response to the ATA IDENTIFY command. The number that is specified by the **Cylinders** registry value is used instead.

Heads
: A value of *hh* causes the ATADISK driver to not rely on the number of heads that is reported by the ATA device in response to the ATA IDENTIFY command. The number that is specified by the **Heads** registry value is used instead.

Sectors
: A value of *ss* causes the ATADISK driver to not rely on the number of sectors per track that is reported by the ATA device in response to the ATA IDENTIFY command. The number that is specified by the **Sectors** registry value is used instead.

CHSMode
: A value of 1 forces the ATADISK driver to use Cylinder/Head/Sector (CHS) addressing mode. If the **CHSMode** value is 0 or the value is not present, the ATADISK driver uses the addressing mode that is reported by the ATA device in response to the ATA IDENTIFY command. The ATADISK driver uses logical block address (LBA) mode, when available.

OEM Considerations for Linear Flash Memory

The following sections describe aspects of linear flash memory that OEMs should be aware of when implementing Windows CE–based platforms and block device drivers for those memory technologies. The sections include information on how an OEM can customize the TrueFFS driver on a Windows CE–based platform. These sections do not contain information for IHVs who want to create applications that use linear flash memory. For this type of information, see "Programming Considerations for Linear Flash Memory."

TrueFFS Driver Customization

The Microsoft Windows CE Platform Builder contains a core library, Tffscore.lib, and a single source file, Flcustom.c, which together compose the TrueFFS driver. Flcustom.c contains translation layers, socket interfaces, and MTDs. All three of these components can be modified to conform to individual design features of a Windows CE–based platform.

However, to integrate any of these three components into the TrueFFS driver, the component's registration routine must be called during initialization of the driver. This task is performed by the **flRegisterComponents** function, which is defined in Flcustom.c. Prototypes for all the components' registration routines are provided in the Stdcomp.h header file.

OEMs can modify **flRegisterComponents** to include the desired subset of components for the application. The **flRegisterComponents** function takes no parameters and has no return value. It simply consists of calls to each component's registration routine.

The following sections provide additional information about the components—MTDs, translation layers, and socket interfaces—and their registration routines.

Memory technology drivers

The TrueFFS driver accesses various types of linear flash memory hardware through MTDs. Each type of linear flash memory requires its own MTD because individual MTDs are tailored to interact with specific linear flash memory chips. For this reason, OEMs can omit any MTDs that are superflous. For example, if a platform supports only built-in DiskOnChip hardware, an OEM can omit the MTDs for Intel linear flash memory chips. By extension, however, OEMs who need to support new types of linear flash memory hardware must create MTDs for that hardware.

The following table shows existing MTDs and their corresponding registration routines.

Manufacturer	Product	Registration routine to be called in Flcustom.c
M-Systems, Inc.	DiskOnChip	flRegisterDOC2000
	Series 2000	flRegisterCDSN
	FlashLite (8-bit mode)	flRegisterI28F008
	FlashLite (16-bit mode)	flRegisterI28F016
Intel	Series II (8-bit mode)	flRegisterI28F008
	Series II (16-bit mode)	flRegisterI28F016
	Series II+	flRegisterI28F016
	Series 100 MiniCard (8-bit mode)	flRegisterI28F008

Manufacturer	Product	Registration routine to be called in Flcustom.c
Intel *(continued)*	Series 100 MiniCard (16-bit mode)	flRegisterI28F016
	Series 200 MiniCard	flRegisterCFISCS
	Value Series 100 (8-bit mode)	flRegisterI28F008
	Value Series 100 (16-bit mode)	flRegisterI28F016
	Value Series 200 (8-bit mode)	flRegisterCFISCS
	Value Series 200 (16-bit mode)	flRegisterCFISCS

Translation layers

Translation layers provide the necessary mapping between FAT file system data structures and the underlying data format on the linear flash memory hardware. The most widely known and used translation layer is the flash translation layer (FTL), a PC Card industry standard for **NOR**-based linear flash memory hardware. A separate implementation of the FTL for **NAND**-based linear flash memory is called the NFTL translation layer. DiskOnChip and other Series-2000 products are **NAND**-based products.

OEMs can omit unnecessary translation layers. For example, an OEM who wants to use only built-in DiskOnChip hardware can omit the FTL and keep the NTFL. However, TrueFFS supports storage of both FAT16 and FAT32 data structures.

The following table shows the registration routines of existing translation layers.

Registration routine	Hardware
flRegisterFTL	**NOR**-based linear flash memory hardware
flRegisterNFTL	**NAND**-based linear flash memory hardware

Socket interfaces

Socket interface components control the access of the TrueFFS driver to sockets in which linear flash memory hardware resides. For PC Cards, the interface components consist of the PC Card socket API. For built-in DiskOnChip devices, the socket interface is a custom software component. An OEM who provides DiskOnChip hardware but not PC Card sockets needs only the DiskOnChip socket layer.

The following table shows the registration routines for existing socket layers.

Registration routine	Hardware
flRegisterFixedFlash	Built-in DiskOnChip hardware
flRegisterCS	PC Card–based linear flash memory hardware

Programming Considerations for Linear Flash Memory

The following sections describe programming aspects of linear flash memory performance on Windows CE that will be useful to most OEMs and IHVs. However, OEMs who want to customize the TrueFFS driver for a Windows CE–based platform should refer to "OEM Considerations for Linear Flash Memory."

Writing a Linear Flash Memory Driver

Although Windows CE provides a linear flash memory driver in the form of the TrueFFS driver, OEMs or IHVs can write their own linear flash memory drivers on Windows CE, if desired.

In general, linear flash memory drivers are stream interface drivers; therefore, they expose stream interface functions that are common to all drivers that are controlled by the Device Manager. For more information about stream interface drivers and stream interface functions, see "Developing Stream Interface Device Drivers."

DSK_IOControl is the main function in the stream interface group that handles all I/O requests. The I/O control codes themselves are the same as those used by the FAT file system. For details about the semantics of those codes, see the Microsoft Windows CE Platform SDK.

A linear flash memory driver is not required to be a stream interface driver. An OEM can implement a custom driver—for example, a monolithic driver—that supports linear flash media as long as the driver fits the Windows CE–based device driver model.

Persistent Database Storage on Block Devices

Linear flash memory can store and update database information, including information contained in the Windows CE Contacts, Tasks, and Calendar databases. Windows CE 2.10 supports databases on any mounted file system, regardless of the file system's storage mechanism. A new database API enables developers to create and mount an existing database volume—for example, the volume contained in the object store—on an external storage device, such as a linear flash memory card.

Operations on a new database volume are identical to operations on an object store database. A new database volume contains both the data and an integrity log that tracks changes for atomic operations. There are several database functions to facilitate these operations, such as the **CeMountDBVol** function, which is used to mount a database. There are also extended versions of existing functions, such as the **CeOpenDatabaseEx** function, which opens a database on a mounted volume. For full documentation on the database API, see the Microsoft Windows CE Platform SDK.

Persistent Registry Storage on Block Devices

In addition to database information, linear flash memory cards can store registry information. However, Windows CE cannot use registry information that is stored in linear flash memory directly. The information must first be copied to the RAM-based registry.

Windows CE versions 2.0 and earlier implement the registry as a RAM-based heap file. However, this means that registry information is lost if power is lost to the Windows CE–based platform's RAM, a circumstance that forces Windows CE to reload the registry from ROM. Windows CE 2.10 provides new functions for saving and restoring registry information to and from any storage location, such as linear flash memory devices. The registry is saved to nonvolatile linear flash memory when the Windows CE–based platform is turned off and restored when power is restored. The **RegCopyFile** function saves registry information, and the **RegRestoreFile** function loads registry information. For full documentation on these two functions, see the Microsoft Windows CE Platform SDK.

One solution for restoring registry information involves a dual-startup setup of the Windows CE–based platform. In this case, the cold startup restores the registry, and a warm startup turns on the system. A warm startup maintains RAM, which means that the restored registry information is available to control the startup sequence. OEMs can provide a special registry cold-startup tool to copy saved registry information from linear flash memory into RAM after the Device Manager loads the TrueFFS. The utility then forces a warm startup of the system.

A single-startup solution is possible if the system uses the internal ROM file system code to restore the registry from its saved location. This is accomplished by using the **WriteRegistryToOEM** and **ReadRegistryFromOEM** functions in the Windows CE–based platform's OEM adaptation layer (OAL). Implementing these two functions can be complex because, at the time they are called, very little system support is present. The Device Manager has not yet been loaded, and no device drivers are available. Therefore, these functions need enough understanding of the external registry storage mechanism to locate and load the registry information without the help of the TrueFFS device driver. For full documentation of **WriteRegistryToOEM** and **ReadRegistryFromOEM**, see the Platform Builder.

Execute-in-Place Functionality

Execute-in-place (XIP) functionality is the ability to run an application directly from linear flash memory, instead of copying the application into RAM memory and running it from there. To support XIP, a device must be linear in nature, meaning it can be mapped into a memory window and read directly through the window as if the linear flash memory were RAM or ROM. Among linear flash memory devices, only **NOR**-based devices, such as minicards or resident flash arrays, can be used for XIP. **NAND**-based devices, such as DiskOnChip do not support XIP because they use complicated access methods to access memory cells within the device. For example, the electrical specifications of DiskOnChip match those of disk drives; disk drives are block devices, and therefore cannot be used to read individual addresses.

Implementing XIP from a built-in linear flash memory device does not require use of the TrueFFS driver. It is required only to install a file system on a linear flash memory device and access it as a random-access storage device. XIP does not need a file system; therefore, implementing XIP does not require use of the TrueFFS driver. An OEM can implement XIP if the target platform provides built-in linear flash memory, such as a resident flash array, or PC Card–based linear flash memory that can be mapped into the system's address space.

OEMs should consider the following factors when implementing XIP from a linear flash memory device:

- Windows CE 2.10 supports XIP from only one address space, the one in which the kernel itself runs. This is because Windows CE modifies the addresses of many system calls, thus enabling the calls to run in the context of the calling thread. Doing so has advantages for memory use, but it limits XIP to the virtual memory address space of the Windows CE kernel.
- Because XIP occurs only in the kernel's address space and Windows CE itself is executed in place, any linear flash memory that might be used for XIP must be built in, or it must be a linear flash memory device that is memory-mapped directly onto the kernel's address space.

For additional information about implementation, see "Starting Windows CE from Linear Flash Memory."

Partitioning Linear Flash Memory

OEMS have the option of configuring linear flash memory as either a single contiguous space or as multiple partitions, even though Windows CE currently supports only single partition. The following is information about partitioning a device for both single and multiple configurations:

- Single partition

  An OEM can treat linear flash memory as a single contiguous address space. In this configuration, linear flash memory can be used either for XIP or for data storage. Data storage can include one or more of the following: a registry, a database, or user data files that are managed by the FAT file system.

- Multiple partitions

  Windows CE 2.10 does not support multiple partitions at this time; however, multiple partitioning enables OEMs to logically divide linear flash memory into separate partitions. In a configuration known as Code Plus Data, one partition is used for XIP and another is used for data storage. Another configuration, known as Data Plus Data, divides linear flash memory into two data storage partitions: one partition provides data storage that is managed by the TrueFFS driver, and the other partition provides data storage that is managed by OEM-defined code.

- Partitions on a DiskOnChip device

  OEMs who use the DiskOnChip device have the option of making the space on the device appear as separate partitions, although Windows CE 2.10 does not provide built-in support for multiple partitions on a single linear flash memory device. M-Systems, Inc., does offer a customization toolkit enabling OEMs to create a hidden data partition on the device. OEMs should contact M-Systems, Inc., for further assistance on how to access this hidden partition from within Windows CE.

Starting Windows CE from Linear Flash Memory

An OEM can use linear flash memory to store a Windows CE OS image, making it possible to start Windows CE directly from a linear flash memory device, instead of from ROM. Starting the OS image from linear flash memory mandates that the platform also support XIP because Windows CE itself runs in XIP mode. In turn, this means that startup is possible only on built-in linear flash memory that is mapped directly into the system's main address space.

One method for implementing startup involves writing a separate startup-loader application that copies the image to the built-in linear flash memory device, and then executes that image. This method does not involve the use of the TrueFFS driver because the device is accessed directly from the startup-loader application. OEMs are free to implement their own startup solutions on linear flash memory devices. OEMs who want to implement startup from DiskOnChip through the use of the existing TrueFFS driver can contact M-Systems, Inc., for assistance.

Further Information on Using Linear Flash Memory with Windows CE

Consult the following sources for further information on using linear flash memory with Windows CE:

- Microsoft Windows CE Platform SDK

  Information on the FAT file system I/O control codes.

- Microsoft Windows CE Platform Builder version 2.11

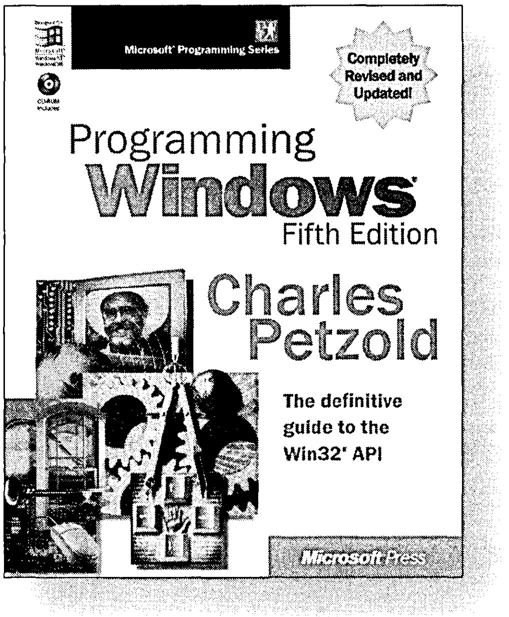
  Information on building custom Windows CE OSs and adapting Windows CE to custom hardware.
- The M-Systems, Inc., Web site

  Details about linear flash memory technologies and software of M-Systems, Inc.
- The Personal Computer Memory Card International Association (PCMCIA) Web site

  Information on the mechanical, electrical, and software specifications for PC Cards.

Index

The *definitive guide* to the **Win32 API**


Microsoft® Programming Series

Completely Revised and Updated!

Programming Windows® Fifth Edition

Charles Petzold

The definitive guide to the Win32® API

Microsoft Press

"Look it up in Petzold" remains the decisive last word in answering questions about Microsoft® Windows® development. And in PROGRAMMING WINDOWS, Fifth Edition, the esteemed Windows Pioneer Award winner revises his classic text with authoritative coverage of the latest versions of the Windows operating system—once again drilling down to the essential API heart of Win32® programming. Packed as always with definitive examples, this newest Petzold delivers the ultimate sourcebook and tutorial for Windows programmers at all levels working with Windows 95, Windows 98, or Windows NT.® No aspiring or experienced developer can afford to be without it.

U.S.A.	**$59.99**
U.K.	£56.49 [V.A.T. included]
Canada	$86.99

ISBN 1-57231-995-X

crosoft Press® products are available worldwide wherever quality computer books are ld. For more information, contact your book or computer retailer, software reseller, or :al Microsoft Sales Office, or visit our Web site at mspress.microsoft.com. To locate your arest source for Microsoft Press products, or to order directly, call 1-800-MSPRESS in ɘ U.S. (in Canada, call 1-800-268-2222).

ices and availability dates are subject to change.

Microsoft®

mspress.microsoft.com

Microsoft® Windows® CE Device Driver Kit

Your official guide to device driver development—direct from the Windows CE team.

Quickly write code that ports the Windows CE operating system to built-in hardware—or almost any conceivable peripheral—with this official Device Driver Kit (DDK). Available for the first time in print, this DDK explains how to implement four models for driver development—native, stream, Universal Serial Bus (USB), and network driver interface specification (NDIS).

Part No. 097-0002205

mspress.microsoft.com

Microsoft*® *Press