Implementing Your Device Driver

Windows Embedded Compact 7 Technical Article

Writers: Mark McLemore

Technical Reviewer: KS Huang, Michael Svob

Published: March 2011

Applies To: Windows Embedded Compact 7

Abstract

This article helps you get started implementing a device driver for Windows Embedded Compact 7. You learn how to create your device driver project from a sample stream driver, add functionality to control your hardware, and implement device driver features that are used by the OS and applications. You should read this article after reading the introductory article Planning Your Device Driver (http://go.microsoft.com/fwlink/?LinkID=210236) and before reading the companion article Building and Testing Your Device Driver (http://go.microsoft.com/fwlink/?LinkID=210199).

Introduction

After you are familiar with the material in the companion article Planning Your Device Driver (http://go.microsoft.com/fwlink/?LinkID=210236), you can begin implementing your device driver. The easiest way to begin this implementation is to start with an existing sample device driver, modify it to fit your design, and add functionality to support your hardware. When you have completed an initial device driver implementation, refer to the article Building and Testing Your Device Driver (http://go.microsoft.com/fwlink/?LinkID=210199) for the steps to build your device driver.

Because each device driver can be very different in architecture, the interfaces it supports, its size, the complexity of its implementation, and the kind of hardware it supports, no single step-by-step procedure for device driver development applies to all device drivers. Your device hardware may generate interrupts or may be a polled device; it may support direct memory access (DMA), and you may be able to turn it on and off for power management support. Applications may require your device driver to manage multiple open instances across multiple devices, or your driver may support only one open instance on one hardware device.

For this reason, this article focuses on the most frequently used scenarios:

- Porting a sample stream driver to start your device driver project.
- Managing device driver context.
- Adding support for interrupt handling.
- Marshaling data between caller and device driver address spaces.
- Adding support for DMA.
- Supporting device interfaces.
- Supporting power management.

Not all implementation tasks in this article may apply to your device driver. Just choose from the topics in this article that apply to your particular device driver design. Each topic provides pointers to Windows Embedded Compact 7 reference documentation that explains the relevant APIs in more detail. Some topics refer you to a sample device driver that provides a working example of the functionality that is explained in the topic; you can study this example code to understand how to implement the functionality you need while you learn how to use the APIs more effectively.

Porting the Sample Stream Driver

Because most Windows Embedded Compact 7 device drivers are stream drivers, Microsoft provides a sample stream driver that you can use to begin developing your device driver. The sample driver code that is described in the following steps does not actually work with specific hardware; you must modify this driver code for it to function correctly with your target device.

1. If you have not already done so, use the integrated development environment (IDE) in Platform Builder to create an OS design. For more information about creating an OS design, see the article Using Platform Builder in Windows Embedded Compact 7 (http://go.microsoft.com/fwlink/?LinkID=210759).

2. Copy the sample stream driver to your OS design. The sample stream driver is located at %_WINCEROOT%\Platform\BSPTemplate\Src\Drivers\Streamdriver.
3. Rename the sample stream driver source files to names that correspond with the functionality you intended for your device driver. You can rename the “SDT” prefix of each function to a prefix that you choose for your device driver.
4. Edit the TARGETNAME and SOURCES values in the sources file so that your driver corresponds with your new file names.
5. Verify that your driver entry points are correctly exposed to the build system. As shown in the sample stream driver source, you should include each function in the .def file and preface each function with an `extern “C”` declaration in the .cpp file.
6. Modify the source code for your driver to implement the stream interface functions. You can start by implementing code to manage open and device contexts. For more information about managing context, see Managing Device Driver Context in this article.
7. Add functionality to control your hardware, including interrupt handling, DMA processing, data marshaling, device interface support, and power management as required. We recommend that you first create header files with constants and data structures that represent your hardware before writing hardware-specific code. Note that adding functionality to control your hardware is typically the bulk of the process to implement your device driver.
8. Add code to connect the functionality you implement in step 7 to the stream interfaces of your driver.
9. If your driver must support additional functionality beyond that which the stream interface functions expose, you can choose to implement custom I/O control codes in the **IOControl** function of your driver.

When you are ready to build your device driver, the article Building and Testing Your Device Driver (http://go.microsoft.com/fwlink/?LinkID=210199) explains how to add registry information for your driver to the Platform.reg file, how to modify Platform.bib and build system files so that the OS design includes your driver, how to build a run-time image with your device driver, and how to test your device driver. After you complete steps 1-5 and have a basic driver shell, verify that you can build your device driver, include it in a run-time image, and access it from a simple test application before adding more functionality in steps 6-9.

Managing Device Driver Context

Device drivers often must manage multiple open instances and multiple devices. When a device driver multiplexes its functionality across multiple open instances or multiple devices, it uses device driver context to keep track of which data structures and system resources go with which open instances and hardware devices. Because device drivers are DLLs with global variables and data structures that are shared by all driver instances, device drivers must use context management techniques to prevent resources that are allocated for one open instance or device from being incorrectly modified or deallocated for another open instance or device. For example, if the device driver deallocates the wrong memory block in response to an **XXX_Close** call, memory leaks, application failures, or an unstable system could occur.

Device drivers use *device context* to manage resources that are associated with a particular hardware device and *open context* to manage resources that are associated with a particular open instance. Depending on the functionality of your hardware, your

device driver might support only one open instance and only one device, multiple open instances on one device, one open instance per hardware device, or multiple open instances across multiple devices. In each case, your device driver must correctly manage each type of context separately.

Managing Device Context

Device drivers typically use device context to manage physical resources and memory areas that are associated with a hardware device. The device context is usually a pointer to a data structure that contains information about a single hardware device and the resources the system uses to control that device. A device driver might use multiple device contexts to support multiple hardware devices. For example, when a network device driver manages multiple network interface cards, it creates and maintains a separate device context for each network interface card.

When the Device Manager calls your **XXX_Init** function, your return value is the device context (a DWORD) that you create for the initialized device. The Device Manager passes a registry path to your driver that describes the device to be initialized, and your driver uses this information to populate any data structures that are associated with the device context that you create for that device.

As an example, the following is a modified version of the stream driver **SDT_Init** function, with additional code to create and return a device context.

```
// SDT_Init
//
// This function initializes the device driver. Its responsibility
// is to set the default state of the device and to allocate and
// initialize any global variables or resources.
//
// Upon success, this function returns a handle to the driver's state
// (device context). This is passed to Open and Deinit when they are
// called.

DWORD SDT_Init(LPCTSTR pContext, DWORD dwBusContext)
{
    BOOL fOk = FALSE;
    DEBUGMSG(ZONE_INIT,
        (TEXT("SDT_Init(): context %s.\r\n"), pContext));

    // Allocate enough storage for this device context:

    SDT_DEV_CONTEXT* pDevContext =
        (SDT_DEV_CONTEXT *)LocalAlloc(LPTR, sizeof(SDT_DEV_CONTEXT));

    // If allocation failed, set the error, clean up, and exit:

    if (pDevContext == NULL)
    {
        SetLastError(ERROR_OUTOFMEMORY);
        DEBUGMSG(ZONE_ERROR,
            (TEXT("SDT_Init(): cannot allocate memory!\r\n")));
        goto cleanup;
    }
```

```
    // Try to read the registry entry for this device into
    // the device context data structure (implement the function
    // ReadRegistry to open the registry key passed in pContext
    // and populate pDevContext with information from this
    // registry key).

    if (!ReadRegistry(pContext, pDevContext))
    {
        DEBUGMSG(ZONE_WARNING,
            (TEXT("SDT_Init(): unable to read registry!")));
        goto cleanup;
    }

    // . . . . . . . . . . . . . . . . . . . . . . . . . . . .
    // Initialize device hardware here
    // . . . . . . . . . . . . . . . . . . . . . . . . . . . .

    fOk = TRUE;

cleanup:
    // If initialization fails, free the device context
    // structure and prepare to return 0 (NULL).
    // Otherwise, if initialization succeeds, return
    // the device context.

    if ( !fOk && pDevContext )
    {
        LocalFree(pDevContext);
        pDevContext = NULL;
    }
    DEBUGMSG(ZONE_INIT,
        (TEXT("SDT_Init(): context %x.\r\n"), pDevContext));
    g_pDevContext = pDevContext;
    return (DWORD) pDevContext;
}
```

In this example, when **SDT_Init** is called, it allocates a structure to hold the device context, reads the registry key for the device (passed in the first parameter), and initializes driver data structures with information from that registry key. After it initializes the device hardware (using information stored in **pDevContext**), it returns **pDevContext** to the caller. If the driver is unable to allocate a device context structure for the device or if it detects an error while initializing hardware, it sets the appropriate error code by calling **SetLastError** and returns 0 (zero) as the value of the device context to indicate that **SDT_Init** did not succeed.

In this example, the device driver supports only one device context. If your device driver supports multiple devices, and therefore, multiple device contexts, you can modify this code to save multiple copies of `pDevContext` in a global device driver data structure such as a linked list. For an example of how to save multiple open contexts in a global device driver data structure, see the section Managing Open Context. You can use this approach to manage multiple device contexts.

Managing Open Context

Device drivers use *open context* to manage multiple open instances on a particular device. When an application calls **CreateFile** for a stream driver, the stream driver creates an open context. The open context is typically a pointer to a data structure that contains information about that open instance. The stream driver uses the open context to associate data pointers and other resources with an open instance.

For example, the following is a modified version of the stream driver **SDT_Open** function with additional code to create and return an open context.

```
// SDT_Open
//
// This function prepares the device driver for reading, writing,
// or both. It creates a context that is returned to an application.
// The application uses this context to call into the driver to
// perform reads and writes.
//
// The driver may optionally allocate a new context each time this
// function is called. This allows multiple applications to use
// the driver simultaneously. The driver is responsible for tracking
// all of the contexts that it creates so that it can deallocate
// them when Close or Deinit are called.
//
DWORD SDT_Open(DWORD hDeviceContext, DWORD AccessCode, DWORD ShareMode)
{
    DEBUGMSG(ZONE_INIT,
        (TEXT("SDT_Open(): device context %x.\r\n"), hDeviceContext));

    // Return NULL if the device context is invalid:

    if ( !hDeviceContext )
    {
        DEBUGMSG(ZONE_OPEN|ZONE_ERROR,
            (TEXT("SDT_Open(): uninitalized device!\r\n")));
        SetLastError(ERROR_INVALID_HANDLE);
        return(NULL);
    }
    SDT_DEVICE_CONTEXT* pDevice = (SDT_DEVICE_CONTEXT *)hDeviceContext;

    //. . . . . .
    // Verify that the hardware can support this
    // additional open instance.
    //. . . . . .

    // Allocate enough storage for this open context:

    SDT_OPEN_CONTEXT* pOpenContext =
        (SDT_OPEN_CONTEXT *)LocalAlloc(LPTR, sizeof(SDT_OPEN_CONTEXT));

    // If allocation fails, set the error and return NULL:

    if (pOpenContext == NULL)
    {
        SetLastError(ERROR_OUTOFMEMORY);
```

```
        DEBUGMSG(ZONE_ERROR,
            (TEXT("SDT_Open(): cannot allocate memory!\r\n")));
        return(NULL);
    }

    //. . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
    // Initialize the pOpenContext structure here and
    // configure the hardware accordingly for this open instance.
    //. . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

    // Use a critical section to protect the linked list of
    // open instances while you insert this instance into the list:

    EnterCriticalSection(&(pDevice->OpenCS));
    InsertHeadList(&pDevice->OpenList, pOpenContext);
    LeaveCriticalSection(&(pDevice->OpenCS));

    // Log a success message and return the open context:

    DEBUGMSG(ZONE_INIT,
        (TEXT("SDT_Open(): open context %x.\r\n"), pOpenContext));
    return (DWORD) pOpenContext;
}
```

In this example, **SDT_Open** first checks that the device context the caller passes into the first parameter is valid. If the device context is not valid, **SDT_Open** signals an error and returns `NULL`. Because **SDT_Init** returns `NULL` if it is unable to create a device context, this validation of the passed-in device context is an important check to include in the **SDT_Open** function; the caller might simply pass the device context that is returned from **SDT_Init** without checking for errors.

After **SDT_Open** verifies that the hardware can support the additional open instance (a code sample is not included here because this verification is hardware-specific), it allocates an SDT_OPEN_CONTEXT structure to store information about the open instance. After configuring the hardware for this open operation and populating this structure with information about the open instance (again, these details are hardware-specific), it saves this open context in a linked list of open instances that are associated with that device. The critical section protects the linked list in case multiple threads call **SDT_Open** simultaneously. After `InsertHeadList` saves the open context in the list, **SDT_Open** returns it to the caller. Of course, you don't have to use a linked list here; you can choose a different data structure to save and locate your open contexts.

Adding Interrupt Handling Functionality

Most peripheral devices generate interrupts to receive service from the OS. Because these peripheral devices can cause or signal interrupts, their device drivers must process interrupts to service their devices. Devices generate interrupts by using physical interrupt request (IRQ) lines. An IRQ line is a hardware line over which a device can send an interrupt signal to the microprocessor. A system interrupt (SYSINTR), which is also known as a logical interrupt, is a mapping of an IRQ as specified by the OEM adaptation layer (OAL). For more information about this mapping and the OAL, see the article The Basics of Bringing up a Hardware Platform (http://go.microsoft.com/fwlink/?LinkId=205801).

A typical Windows Embedded Compact 7 device driver separates interrupt handling into two components:

- An interrupt service routine (ISR)
- An interrupt service thread (IST)

The following sections describe these components in more detail.

Interrupt Service Routines

An ISR is a software routine that hardware invokes in response to an interrupt. ISRs examine an interrupt and determine how to handle it. ISRs handle the interrupt and then return a logical interrupt value. If no further handling is required because the device is disabled or data is buffered, the ISR notifies the kernel with a SYSINTR_NOP value. An ISR must perform very quickly to avoid slowing down the operation of the device and the operation of all lower-priority ISRs. The ISR runs in kernel mode; so it should do the minimum work that is required to service the interrupt. Because the kernel handles the saving and restoring of CPU registers, you can implement your ISR in small and fast C code instead of in assembly code.

The ISR performs the following tasks:

- Determines the source of the interrupt.
- Masks the interrupt.
- Reads data from the device into a software buffer if the data might be lost or another interrupt might overwrite it.
- Clears the interrupt condition on the device.
- Translates the interrupt into a system interrupt (SYSINTR) identifier that the ISR returns to the OS.

Although an ISR might move data from a CPU register or a hardware port into a memory buffer, it typically relies on a dedicated interrupt service thread (the IST) to do most of the required processing. If additional processing is required, the ISR returns a logical interrupt value other than SYSINTR_NOP to the kernel. It then maps a physical interrupt number to a logical interrupt value. For example, the keyboard could be associated with IRQ 4 on one device and IRQ 15 on another device. The ISR, which is in the OAL, translates the hardware-specific value to the Windows Embedded Compact standard value of SYSINTR_KEYBOARD. In this example, SYSINTR_KEYBOARD is the return value from the ISR.

Interrupt Service Threads

When an ISR notifies the kernel of a specific logical interrupt value, the kernel examines an internal table to map the logical interrupt value to an event handle. The kernel wakes the associated IST by signaling the event. This event is a standard synchronization object that wakes a thread in response to an event.

The IST runs in a user mode process called the User Mode Driver Host. For more information about the User Mode Driver Host, see User Mode Driver Framework (http://go.microsoft.com/fwlink/?LinkID=210291) in Windows Embedded Compact 7 Documentation.

The IST performs the following tasks:

- Loads the ISR if the driver uses an installable ISR.
- Creates an event, registers this event in the kernel for the logical interrupt identifier that is associated with the ISR, and waits for the event.
- Performs all interrupt processing when it wakes on the event.

The following section describes how the IST interacts with other interrupt handling components to service an interrupt.

Servicing Interrupts

Figure 1 shows how the interrupt handling components interact with each other and the device hardware during interrupt processing. Note that you implement only some of the components that are shown in this diagram: you typically provide the device driver ISR, platform dependent driver (PDD) lower-layer routines, and OAL routines, and Microsoft provides the kernel exception and interrupt handling functionality in addition to a model device driver (MDD) library that includes the device driver IST.

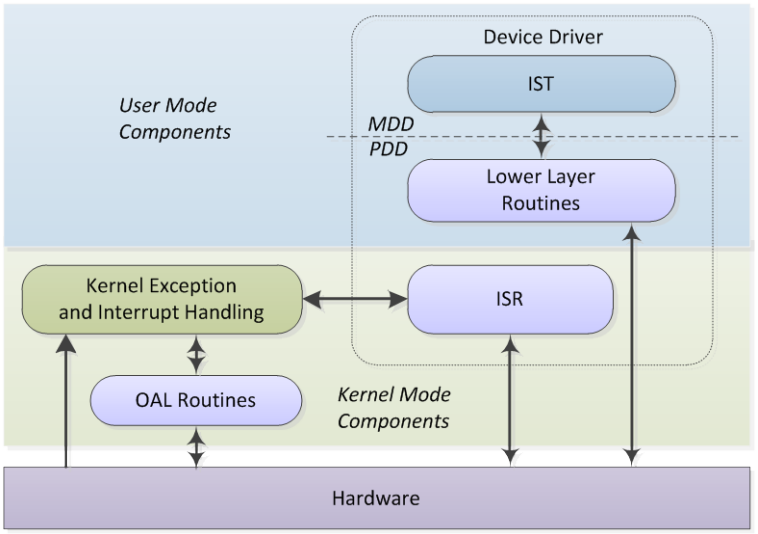

Figure 1 - Interrupt handling components

When an interrupt occurs, interrupt handling typically occurs in the following sequence, although the details can vary depending on the CPU architecture.

1. When your device generates an interrupt, the microprocessor jumps to the kernel exception handler.
2. The exception handler disables all interrupts of an equal or lower priority at the microprocessor and then calls the appropriate ISR for that IRQ. If this IRQ is associated with your device, the exception handler calls your driver ISR.
3. Your ISR services the interrupt. Typically, your ISR also masks the board-level device interrupt so that the OS does not respond to the interrupt signal from your device during interrupt processing.
4. After servicing the interrupt, your ISR returns a logical interrupt identifier to the kernel interrupt handler.

5. The kernel interrupt handler receives the return value from your ISR, reenables all interrupts at the microprocessor except for the current interrupt (which remains masked at the board), and then signals the appropriate IST event.
6. The OS schedules the IST.
7. When the IST wakes, it works to complete the interrupt processing by calling various lower-level driver routines to access the hardware. This process could include moving data into a device buffer or interpreting status data from the device.
8. When the IST completes interrupt processing, it calls the **InterruptDone** function, which in turn calls the **OEMInterruptDone** function in the OAL.
9. **OEMInterruptDone** reenables (unmasks) the current interrupt.

As described in step 4, your ISR services the device interrupt and translates the interrupt into a SYSINTR. It then passes this logical identifier to the kernel as its return value. Your ISR returns one of the return codes listed in Table 1 to notify the kernel how to handle the interrupt.

Table 1: ISR Return Codes

Code	Description
SYSINTR_NOP	The kernel does nothing.
SYSINTR_*XXX*	The kernel schedules interrupt processing so that the IST wakes and does its work. *XXX* specifies the logical interrupt value for the specific interrupt source.
SYSINTR_RESCHED	The kernel reschedules the IST.

Each IRQ is associated with an ISR, and an ISR can respond to multiple IRQ sources. When interrupts are enabled and an interrupt occurs, the kernel calls the registered ISR for that interrupt. When finished, the ISR returns an interrupt identifier. The kernel examines the returned interrupt identifier and sets the associated event. When the kernel sets the event, the IST starts processing.

Registering Your Interrupt Service Routine

Your driver must register its ISR with the kernel unless the driver relies on the OAL ISR function to handle its interrupt. The driver must register its ISR with the kernel so that the kernel calls the ISR when its associated physical interrupt occurs. If your driver does not register an ISR, a default ISR, which is installed by the OAL in **OEMInit**, handles any interrupts that the device generates.

If your device driver has an ISR, your device driver must perform the following actions when it loads:

1. Register its ISR with the kernel.
2. Map the IRQ of the device to a unique SYSINTR.

To associate your ISR with your device, you register your ISR with the exception handler when you start the OS by including a call to **HookInterrupt** in **OEMInit**. To map an IRQ to a SYSINTR, you call **OALIntrStaticTranslate** in **OEMInit**. During boot, the kernel calls **OEMInit** in the OAL, and then **OEMInit** calls **HookInterrupt** to inform the exception handler of the address of your ISR and the IRQ that your ISR services.

For more information about interrupt service routine registration, see HookInterrupt (http://go.microsoft.com/fwlink/?LinkID=210361) in Windows Embedded Compact 7 Documentation.

The interrupt handler registration process registers an event that is associated with the system interrupt SYSINTR. When your device driver starts, it creates an IST, and then calls **InterruptInitialize** to register this event. The IST can then use **WaitForSingleObject** to wait on this event and register it with the interrupt handler. You can register your IST for one or more logical interrupts (SYSINTRs).

If you use the Microsoft implementation of the MDD for a particular driver, you do not have to write code to register the interrupt. The MDD layer of the driver registers the driver for interrupts. If you write a monolithic driver, you must implement code to register the IST of the driver with the interrupt handler. To do this, use the **CreateEvent** function to create an event, and then use the **InterruptInitialize** function to associate the event with a SYSINTR.

If your device driver must stop processing an interrupt, call the **InterruptDisable** function. When your 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. If necessary, you can register for the interrupt again.

Handling Nested Interrupts

To prevent loss and delay of high-priority interrupts, the Windows Embedded Compact kernel uses *nested interrupts*. Nested interrupts allow interrupt requests (IRQs) of a higher priority to preempt IRQs of a lower priority. Because of this preemption at the hardware level, ISRs of a higher priority might preempt ISRs of a lower priority.

The following steps explain how Windows Embedded Compact handles nested ISR calls:

1. The kernel disables all other IRQs that have the same priority or lower priority as the IRQ that invoked the current ISR call.
2. If a higher-priority IRQ arrives before the current ISR completes processing, the kernel saves the current ISR state. The state includes only the set of CPU registers that the OS supports for use in an ISR.
3. The kernel calls the higher-priority ISR to handle the new request.
4. The kernel loads the original ISR state and continues processing.

The kernel saves the state of the currently running ISR when a higher-priority interrupt occurs and restores it after the high-priority ISR has finished processing. In most cases, a preempted ISR does not detect that it has been preempted. The level of interrupt nesting is limited solely by what the hardware platform can support.

An ISR that uses too much time can cause other interrupts to be missed entirely, resulting in erratic or sluggish performance of the whole system. By separating interrupt processing into a very short ISR and a longer IST, ISRs can mask interrupts for as little time as possible. ISRs can only mask interrupts that are of equal or lower priority. Windows Embedded Compact can nest as many ISRs as the hardware platform supports. Other than a possible delay in completion, a higher-priority interrupt does not affect the processing of a running ISR.

Handling Shared Interrupts

On some hardware platforms, multiple devices might share a single interrupt request line. Sharing is necessary because hardware platforms have a finite number of inputs. When interrupts are shared, the microprocessor is aware of an interruption, but it is not necessarily aware of which device interrupted it. To determine this, the OS exception handler must typically examine device-specific registers that indicate the state of a specified device interrupt line.

You can register routines with the kernel that are automatically invoked when a shared interrupt occurs. The ISR that you hook to an interrupt in **OEMInit** must call **NKCallIntChain**, which is a kernel function, to examine a list of installed ISRs for the interrupt that has been signaled.

The following sequence of steps describes how the OS handles shared interrupts:

1. Your ISR routine calls **NKCallIntChain** to examine a list of installed ISRs for the interrupt that a device has signaled.
2. If the first ISR in the list determines that its associated device has asserted the interrupt, it performs any necessary work and then returns the SYSINTR that is mapped to the interrupt. If this ISR determines that it is not necessary for the IST to do additional processing, it returns SYSINTR_NOP.
3. If the first ISR in the list determines that its associated device has not asserted the interrupt, it returns SYSINTR_CHAIN, which causes **NKCallIntChain** to call the next ISR in the chain.

The order of installing ISRs is important because it sets priority. The first ISR on the chain has priority over the succeeding ISRs on the chain. For more information about handling shared interrupts, see NKCallIntChain (http://go.microsoft.com/fwlink/?LinkID=210362) in Windows Embedded Compact 7 Documentation.

Interrupt Handling Functions

The following table lists the Windows Embedded Compact 7 functions that device driver developers typically use to implement device driver interrupt handling functionality.

Table 2: Interrupt Handler Functions

Function	Description
InterruptDisable	Disables a hardware interrupt that its interrupt handler specifies.
InterruptDone	Signals to the kernel that interrupt processing is complete.
InterruptInitialize	Initializes a hardware interrupt with the kernel. This initialization allows the device driver to register an event and enable the interrupt.
HookInterrupt	Associates an ISR with an IRQ. You call this function from within OEMInit to make this association.

Function	Description
UnhookInterrupt	Breaks the association between an ISR and an IRQ.
CreateEvent	Creates a named or unnamed event object.
SetInterruptEvent	Allows a device driver to cause an artificial interrupt event.
SetEvent	Sets the state of the specified event object to signaled.
ResetEvent	Sets the state of the specified event object to nonsignaled.
PulseEvent	Provides a single operation that sets the state of the specified event object to signaled, and then resets it to nonsignaled after releasing the appropriate number of waiting threads.
WaitForSingleObject	Returns when the specified object is in the signaled state or when the time-out interval elapses. The kernel provides special treatment for interrupt events. ISTs cannot use WaitForMultipleObject with an interrupt event.
CeSetPowerOnEvent	Signals events during suspend or resume. Power handlers call this function.

For more information about these functions, see the reference information for each function in Windows Embedded Compact 7 Documentation.

Marshaling Data Between Memory Buffers

In Windows Embedded Compact 7, each process has its own virtual memory space and memory context. When an application moves data to or from your device driver, you must *marshal* the data between the address space of the application and the address space of your device driver. You marshal data in one of two ways:

- Copy the data between the application memory buffer and a memory buffer that your device driver manages.
- Map the virtual memory address of the application memory buffer to a physical memory location so that your device driver can directly access the application memory buffer without making an extra copy.

You can use the kernel buffer marshaling functions in Table 3 to marshal data between an application and your device driver.

Table 3: Kernel Buffer Marshaling Functions

Function	Description
CeOpenCallerBuffer	Checks access and marshals a buffer pointer from the source process so that the current process can access it.
CeCloseCallerBuffer	Frees any resources that CeOpenCallerBuffer allocated.

Function	Description
CeAllocAsynchronousBuffer	Remarshals a buffer that CeOpenCallerBuffer already marshaled so that the server can use it asynchronously after returning from the API call.
CeAllocDuplicateBuffer	Abstracts the work that is required to make secure copies of the API parameters.
CeFreeAsynchronousBuffer	Frees any resources that CeAllocAsynchronousBuffer allocated.
CeFreeDuplicateBuffer	Frees a duplicate buffer that CeAllocDuplicateBuffer allocated.
CeFlushAsynchronousBuffer	Flushes any changed data between the source and the destination buffer that CeAllocAsynchronousBuffer allocated.

Keep in mind that your device driver must explicitly map all the pointers that the data structures contain. For example, **DeviceIoControl** buffers are often structures that contain data, and some of this data could be pointers to other data structures. For more information about buffer marshaling functions, see Kernel Buffer Marshaling Reference (http://go.microsoft.com/fwlink/?LinkID=210258) in Windows Embedded Compact 7 Documentation.

Synchronous and Asynchronous Access

In Windows Embedded Compact 7, marshaling depends on whether a pointer is used synchronously or asynchronously. If a pointer parameter or embedded pointer is used synchronously, the address space of the calling process is accessible for the duration of a call to the driver. This availability eliminates the need for a copy. The driver can use, unchanged, the calling process pointer, which can then directly access the memory of the caller. This is the *direct access* method of marshaling.

If your driver uses a pointer asynchronously, it is critical that the caller memory buffer be accessible when the address space of the caller is unavailable. Windows Embedded Compact 7 includes the **CeAllocAsynchronousBuffer** and **CeFreeAsynchronousBuffer** functions for drivers to marshal pointer parameters and embedded pointers when your driver needs asynchronous access to a caller memory buffer. When a thread like an IST requires access to the caller buffer, these marshaling helper functions choose between copying the data or using a buffer aliasing mechanism to avoid copying data. This choice is dependent on the size of the buffer involved. If the buffer is small enough, the kernel duplicates the buffer by copying the data. However, this duplication can affect performance, and if the buffer is too large to duplicate, the kernel creates an alias for the buffer instead of copying the data.

The following code is an example of marshaling pointers for asynchronous access in a device driver **IOControl** function.

```
// In XXX_IOControl
//
// CeOpenCallerBuffer generates g_pMappedEmbedded
```

```
// . . .

hr = CeAllocAsynchronousBuffer((PVOID *) &g_pMarshaled,
            g_pMappedEmbedded, pInput->dwSize, ARG_I_PTR);

// Check for FAILED(hr) == true, handle the error.
// . . .
// Use the pointer
// . . .
// When done with the pointer:

Hr = CeFreeAsynchronousBuffer((PVOID)g_pMarshaled,
            g_pMappedEmbedded, pInput->dwSize, ARG_I_PTR);

// Call CeCloseCallerBuffer . . .
```

For more information about asynchronous access, see CeAllocAsynchronousBuffer (http://go.microsoft.com/fwlink/?LinkID=210363) in Windows Embedded Compact 7 Documentation.

Supporting Direct Memory Access

Direct memory access (DMA) is a method that you can use to transfer data from a device to memory, from memory to a device, or from memory to memory, without the help of the microprocessor. In a *standard DMA transfer*, a DMA controller performs the transfer. In a *bus master transfer*, a peripheral performs the transfer. *Common* buffer DMA operations use a contiguous buffer in main memory, while *scatter/gather* DMA uses multiple blocks at different memory addresses. The type of DMA transfer you choose depends on the design of your hardware device. You can perform common buffer and scatter/gather DMA operations by using CEDDK.dll DMA functions or by directly calling kernel functions.

In Windows Embedded Compact 7, you can perform DMA in three ways:

- By using common DMA functions such as **HalAllocateCommandBuffer**, **HalFreeCommonBuffer**, and **HalTranslateSystemAddress**.
- By using DMA abstraction functions.
- By directly calling kernel functions, such as **AllocPhysMem** and **FreePhysMem.**

Which method you use depends on your device, the type of memory alignment it uses, whether it supports scatter/gather DMA, and whether you implement your driver for bus independence. The following subsections explain each of these methods in more detail.

Common DMA Functions

The following table lists three CEDDK.dll functions that are useful for common buffer DMA transfers. These functions handle bus and hardware platform-specific address translations for you; they also handle address translations between the system and the PCI bus for the DMA controller to use for copying data.

Table 4: Common DMA Functions

Function	Description
HalAllocateCommonBuffer	Allocates memory, locks it down, and maps it so that the microprocessor and the device can access it simultaneously.
HalFreeCommonBuffer	Frees a common buffer that is allocated by **HalAllocateCommonBuffer**, together with all the resources that the buffer uses.
HalTranslateSystemAddress	Translates a physical system address to a logical bus address, which you can pass to a bus controller for DMA operations.

These functions translate a physical RAM address to the corresponding bus-relative physical address for the DMA controller. To set up a common buffer for bus master DMA by using the CEDDK.dll functions, a bus master DMA device driver can call **HalAllocateCommonBuffer**, passing in a DMA_ADAPTER_OBJECT structure that contains information about the DMA adapter.

The following code example shows a call to **HalAllocateCommonBuffer** to allocate a physical memory buffer address for DMA.

```
// Set up the DMA adapter descriptor structure.

DMA_ADAPTER_OBJECT AdapterObject;
AdapterObject.ObjectSize = sizeof(AdapterObject);
AdapterObject.InterfaceType = Internal;
AdapterObject.BusNumber = 0;

// Allocate a physical buffer.

m_vuaBuf = (PBYTE)HalAllocateCommonBuffer(&AdapterObject,
                cbBufSize, &m_paBuf, FALSE);

// Unable to allocate a physical buffer.

if (m_vuaBuf == NULL)
{
    RETAILMSG(1,(L"unable to allocate %d bytes\r\n", cbBufSize));
    ret = ERROR_OUTOFMEMORY;
    ASSERT(0);
    HalFreeCommonBuffer(NULL, 0, m_paBuf, m_vuaBuf, FALSE);
    goto cleanup;
}

// . . .
// Use the buffer
// . . .
```

```
// When done with the buffer:

HalFreeCommonBuffer(&AdapterObject, cbBufSize, m_paBuf,
    m_vuaBuf, FALSE);
```

In this example, `m_vuaBuf` is the DMA buffer virtual uncached address, `cbBufSize` is the buffer length, in bytes, and `m_paBuf` is the buffer physical address. The last parameter is set to FALSE to disable caching. When **HalAllocateCommonBuffer** successfully returns, it allocates a shared buffer of locked, physically contiguous pages that the microprocessor and the device can access simultaneously for DMA operations.

For an example that uses these common DMA functions for scatter/gather DMA, see the sample ATAPI device driver source that is located in %_WINCEROOT%\Public\Common\Oak\Drivers\Block\ATAPI. Note that to set up scatter/gather DMA, you must simultaneously use multiple pairs of base addresses and lengths as shown in the sample driver.

For more information about common DMA functions, see Ceddk.dll DMA Functions (http://go.microsoft.com/fwlink/?LinkID=210250) in Windows Embedded Compact 7 Documentation.

DMA Abstraction Layer

Windows Embedded Compact 7 includes a DMA abstraction library with which you can implement DMA support in a bus-independent manner. The purpose of the DMA abstraction library is to create an interface with which a device driver developer can implement DMA support, and also to provide an interface for platform-dependent DMA handler routines. By using the DMA abstraction library functions you can use an onboard or built-in DMA controller through a chip-specific DMA driver that works with a standard set of bus-independent DMA functions.

The DMA abstraction framework is made up of three components:

- The DMA model device driver (MDD).
- The DMA platform dependent driver (PDD).
- The DMA abstraction functions of CEDDK.dll.

CEDDK.dll forwards all function requests to the DMA driver by using IOCTLs. Table 5 lists the CEDDK.dll functions that are included in the DMA abstraction layer.

Table 5: DMA Abstraction Layer Functions

Function	Description
DMAAllocateChannel	Prepares the system for a DMA operation on behalf of the target device.
DMAFreeChannel	Frees a DMA channel buffer that DMAAllocateChannel allocated, together with all the resources that the DMA channel uses to transfer data.

Function	Description
DMAOpenBuffer	Maps a virtual buffer to a physical DMA block.
DMAGetAdapter	Returns the length and physical address of a DMA buffer block that DMAOpenBuffer creates.
DMAGetBufferPhysAddr	Returns the length and physical address of a DMA buffer block that DMAOpenBuffer creates to map a virtual buffer to physical memory.
DMAFlushBlockBuffer	Manually flushes the DMA buffer block to regain cache coherency.
DMACloseBuffer	Frees all resources that DMAOpenBuffer allocates to map a virtual buffer to a physical DMA block.
DMAStartTransfer	Puts a DMA transfer into auto start mode.
DMAIssueMultipleBufferTransfer	Queues multiple DMA transfer requests.
DMAIssueRawTransfer	Sets up map descriptor registers for a channel to map a DMA channel from a locked buffer. This function fails if other DMA transfers are queued in the DMA channel.
DMAIssueTransfer	Sets up map descriptor registers for a channel to map a DMA transfer from a locked buffer. If other DMA transfers are queued in the DMA channel, this function queues this transfer.
DMARawTransferControl	Directly controls hardware-mapped DMA transfers, unless the transfer is already completed or is not queued.
DMATransferOnBlocks	Starts a subordinate DMA transfer that uses an associated memory block.
DMAGetStatus	Gets the current active or queued DMA transfer status.
DMAGetContexts	Retrieves DMA transfer object context values.
DMACancelTransfer	Cancels an active DMA transfer.

Function	Description
DMACloseTransfer	Closes a DMA transfer and releases all related resources.

For more information about these functions, see the reference information for each function in Windows Embedded Compact 7 Documentation.

Direct Calls to Kernel Functions

You can make direct calls to kernel functions to allocate physical memory for DMA to use. Direct calls are useful when you need a finer degree of control over DMA operations than is possible by using the DMA functions in CEDDK.dll. For example, by using kernel functions you can change the default memory alignment of a DMA transfer. Because CEDDK.dll functions use a default memory alignment of 64 KB, you may need to use direct kernel calls if your device requires a smaller memory alignment. When you use direct calls to kernel functions to manage DMA transfers, you must handle any platform-specific address translations. You can call **HalTranslateSystemAddress** to translate these addresses.

Table 6 lists the kernel functions that you call to allocate and deallocate a physical memory buffer for DMA transfers.

Table 6: Kernel DMA Buffer Functions

Function	Description
AllocPhysMem	Allocates a section of physically contiguous memory.
FreePhysMem	Releases physical memory back to the system.

For a DMA example that uses these kernel functions, see the sample device driver source that is located in %_WINCEROOT%\Public\Common\Oak\Drivers\USB\Hcd.

Supporting Device Interfaces

Device interfaces are the methods that applications use to access device driver capabilities. Because a device driver can implement any arbitrary set of commands in its **IOControl** function, it is useful to have a standard set of IOCTL commands that device drivers can implement for common operations. The existence of a standard set of IOCTL commands makes it possible for applications to use the same IOCTL command for an operation that is supported by multiple device drivers. For example, Windows Embedded Compact Power Manager uses a set of common device interfaces to issue power management commands to device drivers. For more information about power management, see the section Supporting Power Management in this article.

Device interface classes are a way to organize and describe these common IOCTL commands. Device drivers use device interface classes to implement device interfaces, and a device driver can have zero or more device interface classes. Platform Builder includes files that uniquely identify each interface class with a GUID; the header (.h) file that declares the interface typically defines the GUID and associates the GUID with the interface. The following code example shows how a header file typically defines a device interface GUID:

```
#define DEVCLASS_IFCNAME_STRING \
        TEXT("{12345678-1234-1234-1234567887654321}")

#define DEVCLASS_IFCNAME_GUID \
        { 0x12345678, 0x1234, 0x1234, \
            { 0x12, 0x34, 0x56, 0x78, 0x87, 0x65, 0x43, 0x21 }}
```

Table 7 lists some of the predefined interfaces and the header file under %_WINCEROOT% that defines each interface.

Table 7: Device Interfaces

Interface	Header file
BATTERY_DRIVER_CLASS	Public\Common\OAK\Inc\Battery.h
BLOCK_DRIVER_GUID	Public\Common\SDK\Inc\Storemgr.h
CDDA_MOUNT_GUID	Public\Common\SDK\Inc\Storemgr.h
CDFS_MOUNT_GUID	Public\Common\SDK\Inc\Storemgr.h
DEVCLASS_CARDSERV_GUID	Public\Common\SDK\Inc\Cardserv.h
DEVCLASS_DISPLAY_GUID	Public\Common\SDK\Inc\Winddi.h
DEVCLASS_KEYBOARD_GUID	Public\Common\SDK\Inc\Keybd.h
DEVCLASS_STREAM_GUID	Public\Common\SDK\Inc\Pnp.h
FATFS_MOUNT_GUID	Public\Common\SDK\Inc\Storemgr.h
FSD_MOUNT_GUID	Public\Common\SDK\Inc\Storemgr.h
NLED_DRIVER_CLASS	Public\Common\SDK\Inc\Storemgr.h
PMCLASS_BLOCK_DEVICE	Public\Common\SDK\Inc\Pm.h
PMCLASS_DISPLAY	Public\Common\SDK\Inc\Pm.h
PMCLASS_GENERIC_DEVICE	Public\Common\SDK\Inc\Pm.h
PMCLASS_NDIS_MINIPORT	Public\Common\SDK\Inc\Pm.h
STORE_MOUNT_GUID	Public\Common\SDK\Inc\Storemgr.h
STOREMGR_DRIVER_GUID	Public\Common\SDK\Inc\Storemgr.h
UDFS_MOUNT_GUID	Public\Common\SDK\Inc\Storemgr.h

Note You are not restricted to this list of predefined interface classes. You can define your own interface classes.

Advertising a Device Interface

When a device driver supports one or more device interfaces, it must notify the OS of the device interfaces that it supports. Your device driver should export a device interface only if it entirely implements that device interface. Any application or OS component that interacts with your driver expects full support of the device interfaces that you advertise.

You can advertise a device interface in one of two ways:

- By defining a registry subkey called **IClass** that specifies one or more GUIDs that represent the device interfaces that your driver supports.
- By calling the Device Manager function **AdvertiseInterface** to announce the interfaces that your device driver exposes.

Typically, you advertise with **AdvertiseInterface** when you want to explicitly send notifications for a removable-media storage device or when your device driver discovers which interfaces it is to expose to the OS and applications. When you call **AdvertiseInterface**, your announcement is available to any application or OS component that calls **RequestDeviceNotifications**. These announcement recipients can call **StopDeviceNotifications** to stop receiving device interface notifications.

For more information about **AdvertiseInterface**, see AdvertiseInterface (http://go.microsoft.com/fwlink/?LinkID=210985) in Windows Embedded Compact 7 Documentation. For more information about device interface notifications and **IClass**, see Device Interface Notifications (http://go.microsoft.com/fwlink/?LinkID=210251) in Windows Embedded Compact 7 Documentation.

Supporting Power Management

Power Manager is a system software component that manages device power to improve overall OS power efficiency. For more information about power management in Windows Embedded Compact 7, see Power Management (http://go.microsoft.com/fwlink/?LinkID=210272) in Windows Embedded Compact 7 Documentation.

You can add power management support to your device driver to reduce power consumption on your target platform. To do this, your driver must support the stream interface. For more information about the stream interface, see the companion article Planning Your Device Driver (http://go.microsoft.com/fwlink/?LinkID=210236).

After you implement the stream interface in your device driver, you can add power management support by adding power management IOCTL functionality to your driver, by notifying Power Manager that your driver has power management capability, and by implementing device power states. We recommend that you use power management IOCTLs to manage power to your device instead of using the **XXX_PowerUp** and **XXX_PowerDown** stream interface functions.

Adding Power Management IOCTLs

Power Manager communicates to a device with the IOCTLs listed in Table 8. You implement these IOCTLs in your device driver to respond to calls from Power Manager.

Table 8: Power Management IOCTLs

Function	Description
IOCTL_POWER_CAPABILITIES	Checks device-specific capabilities.
IOCTL_POWER_GET	Gets the current device power state.
IOCTL_POWER_SET	Requests a change from one device power state to another.
IOCTL_REGISTER_POWER_RELATIONSHIP	Notifies the parent device so that the parent device can register all the devices that it controls.

For more information about these IOCTLs, see Power Management I/O Controls (http://go.microsoft.com/fwlink/?LinkID=210273) in Windows Embedded Compact 7 Documentation.

Notifying Power Manager

To notify Power Manager of your device driver power management capabilities, you expose a power management device interface that allows Power Manager to query your device power management capabilities and to control its power state. For more information about device interfaces, see the section Supporting Device Interfaces in this article.

The power management interface you implement depends on the kind of device that your driver supports. Table 9 lists predefined power management interfaces and the kind of device each supports for managing power on that type of device.

Table 9: Power Management Interfaces

Interface	Device type
PMCLASS_BLOCK_DEVICE	Storage device
PMCLASS_DISPLAY	Display device
PMCLASS_NDIS_MINIPORT	Network device
PMCLASS_GENERIC_DEVICE	All other devices

These interfaces are defined in the header file %_WINCEROOT%\Public\Common\SDK\Inc\Pm.h. After you implement the required interface for your device type, call the **AdvertiseInterface** function to advertise your interface. Power Manager uses **IOControl** commands to query your device power management capabilities after you advertise your interface. In addition, to register your device driver for power management notifications, call the **RequestPowerNotifications** function and pass a handle to a message queue that is exclusively created for power management notifications. Do this only if your driver must respond to a power notification and can afford to incur the associated overhead. Typically, after your device driver advertises power management capabilities to Power Manager, your driver needs

to process only **DeviceIoControl** calls from Power Manager. For more information about **RequestPowerNotifications**, see RequestPowerNotifications (http://go.microsoft.com/fwlink/?LinkID=210986) in Windows Embedded Compact 7 Documentation.

Implementing Device Power States

To implement power management functionality, your driver supports transitions between predefined *power states*. Power Manager passes a device power state to your driver; your driver must then map the state to its device capabilities and also perform the applicable state transition on the device.

Table 10 describes the device power states.

Table 10: Device Power States

Device power state	Registry key	Description
Full on	**D0**	The device is on and running. It is receiving full power from the system and is delivering full functionality to the user.
Low on	**D1**	The device is fully functional at a lower power or performance state than D0. The D1 state is applicable when the device is being used; however, peak performance is unnecessary, and power is at a premium.
Standby	**D2**	The device is partially powered and has automatic wake-up on request.
Sleep	**D3**	The device is partially powered and has device-initiated wake-up, if available. A device in state D3 is sleeping but capable of raising the system power state on its own. It consumes only enough power to be able to do this, which must be less than or equal to the amount of power that is used in state D2.
Off	**D4**	The device has no power. A device in state D4 should not be consuming any significant power. Some peripheral buses require static terminations that intrinsically use nonzero power when a device is physically connected to the bus. A device on such a bus can still support D4.

A physical device does not have to support all the device power states that are shown in Table 10. The only device power state that all devices must support is the full-on state, D0. A driver that receives a request to enter a power state that its device hardware does not support enters the next available power state that it supports. For example, if Power Manager requests that your driver enter the D2 state and your device does not support D2, your driver can instead transition the device to the state D3 or

D4. This transition is possible if Power Manager supports one of these states. If Power Manager requests that your device enter the D3 state but your device cannot wake the system, we recommend that your driver enters the D4 state by turning off the power instead of staying in standby mode. These rules are intended to simplify device driver implementation.

Power Manager maps system power states to the corresponding device power states. For example, if a device supports only power states D0 and D4, Power Manager does not immediately request that the device enter the D4 power state when it transitions from the full-on power state. Power Manager waits until the system enters a system power state in which D3 or D4 is configured as the maximum device power state for that device. If D0, D1, or D2 is configured as the maximum power state, Power Manager keeps the device at D0.

When your device driver starts, it puts the device in the full-on state D0. Before your device driver is unloaded, it should put the device into D4 (the power off state), if possible. If your device enters another device power state at startup, it can issue a **DevicePowerNotify** request while processing IOCTL_POWER_CAPABILITIES.

For more information about device power states, see CEDEVICE_POWER_STATE (http://go.microsoft.com/fwlink/?LinkID=210364) in Windows Embedded Compact 7 Documentation. For more information about IOCTL commands that Power Manager issues to your device driver, see Power Management I/O Controls (http://go.microsoft.com/fwlink/?LinkID=210273) in Windows Embedded Compact 7 Documentation.

Conclusion

This article explains the most frequent tasks that are required to implement a Windows Embedded Compact 7 device driver, and provides pointers to sample code and reference documentation to help you develop a working device driver. It helps you learn how to start your device driver project from a sample stream driver. You then add functionality to manage device context and open context when the OS and applications initialize and open your device driver. This article explains how to service interrupts from your device, register your interrupt handler with the OS, and handle shared interrupts. It also helps you learn how to marshal data between address spaces, use DMA to move data to and from your device, advertise device interfaces to the rest of the system, and support power management IOCTL commands. With this knowledge, and the information in companion articles Planning Your Device Driver (http://go.microsoft.com/fwlink/?LinkID=210236) and Building and Testing Your Device Driver (http://go.microsoft.com/fwlink/?LinkID=210199), you can develop a working, full-featured device driver to support your device hardware in Windows Embedded Compact 7.

Additional Resources

- Windows Embedded website (http://go.microsoft.com/fwlink/?LinkID=203338)

Copyright