Bus Driver Development Based on KMDF

October 21, 2008

Abstract

This paper provides information about how to write bus drivers that are based on the kernel-mode driver framework (KMDF) for the Windows® family of operating systems. It describes when developing a new bus driver is appropriate, shows how to implement common features of a bus driver, and provides tips for testing and debugging a bus driver.

The paper assumes that you have experience developing kernel-mode drivers that are based on KMDF. Some familiarity with the Windows Driver Model (WDM) is also helpful.

This information applies for the following operating systems:

- Windows Server® 2008
- Windows Vista®
- Windows Server 2003
- Windows XP
- Windows 2000

References and resources discussed here are listed at the end of this paper.

For the latest information, see:

http://www.microsoft.com/whdc/driver/wdf/KMDFBusDrv.mspx

Document History

Date	Change
October 14, 2008	First publication

Contents

Introduction

A bus driver is a kernel-mode driver that exposes one or more child device objects to represent the devices that are attached to a bus or the functions that a multifunction device supports. Such a driver is typically—but not always—loaded beneath a function driver in the child driver stack. A driver is also considered a bus driver if it creates a "raw" PDO (physical device object) to enable sideband communication with a device or to manage a device without a function driver.

A bus driver can service an actual hardware bus, a hub or adapter that provides several different functions, or even a software component that emulates a device.

This paper uses the following terms to identify the devices and functions:

- The *parent bus* is the hardware bus, hub, adapter, software component, or multifunction device to which a child device or function is attached.
- A *child device* is a device or function that is attached to the parent bus. A PDO represents each child device.
- *Enumeration* is the process of identifying the child devices that are attached to the parent device.

This paper describes how to write a bus driver that is based on the kernel-mode driver framework (KMDF, also called *the framework*). It does not cover developing bus drivers that are based on the Windows® Driver Model (WDM). KMDF offers features that are designed specifically for bus drivers and therefore makes implementing a KMDF bus driver much simpler than implementing the same driver by using WDM. If your driver requires access to underlying WDM features, KMDF provides a way to access WDM. For basic information about KMDF, see "Windows Driver Foundation" on the WHDC Web site and *Developing Drivers with the Windows Driver Foundation* by Penny Orwick and Guy Smith.

This paper assumes that you are familiar with the implementation requirements for KMDF function and filter drivers and with the basic terminology that is related to kernel-mode drivers for Windows.

When to Write a Bus Driver

Microsoft supplies bus drivers for most hardware buses such as USB, IEEE 1394, and PCI, and for devices that comply with class specifications that run over such buses. For example, the USB common class generic parent driver (USBCCGP.sys) supports USB class devices. Third parties rarely are required to write bus drivers for such device types. Consider writing a bus driver only in the following situations:

- You must support a new type of parent bus that none of the existing Microsoft-supplied bus drivers supports.
- You are splitting a single device into several virtual devices that require individual Plug and Play functionality, and you cannot use any of the existing Microsoft-supplied bus drivers.
- You must support sideband communication with your device because a driver that is layered above your driver in the device stack prevents applications or other drivers from communicating with your device or driver.

You should carefully consider whether your situation requires a new bus driver or whether you can provide the required functionality by using a simpler approach. In the following scenarios, developing a new bus driver is not appropriate:

- Your device requires resource arbitration.

  Windows does not expose resource arbitration to drivers. Instead of writing a bus driver, you should use the Microsoft-supplied Mf.sys driver if possible. This driver handles resource assignment and reports resources correctly for both 32-bit and 64-bit platforms. For more information about how to use Mf.sys, see "Using the System-Supplied Multifunction Bus Driver" in the Windows Driver Kit (WDK).

  By default, the Mf.sys driver is not installed on Windows Vista® and later versions, although it is provided as part of the release. For information about how to install this driver and devices that use it, see the KB article "You cannot install a device that requires the Mf.sys device driver in Windows Vista."

- Your device supports several individual functions, but these functions do not have individual Plug and Play or power-management capabilities and do not require individual management in Device Manager.

  In this case, you should consider creating a device object namespace instead of writing a bus driver. A device object namespace enables clients to open the individual device functions and enables multiple clients to have simultaneous access to a given function. For more information, see the driver tip "No time to write a bus driver? Try using a device object namespace" on the WHDC Web site.

Bus Drivers, Device Objects, and the Device Stack

The type and number of device objects that the bus driver creates and the contents of the resulting device stacks are different for a typical bus driver and for a bus driver that creates a raw PDO.

Device Objects and Device Stack for a Typical Bus Driver

One driver typically serves as a function driver for the parent bus in addition to serving as a bus driver for the child devices. In its role as function driver, the driver creates a functional device object (FDO) for the parent bus, and in its role as bus driver, the driver creates a PDO for each attached child device or device function. Figure 1 shows the relationships among the device objects and device stacks for the sample parent Toaster bus and child Toaster devices.

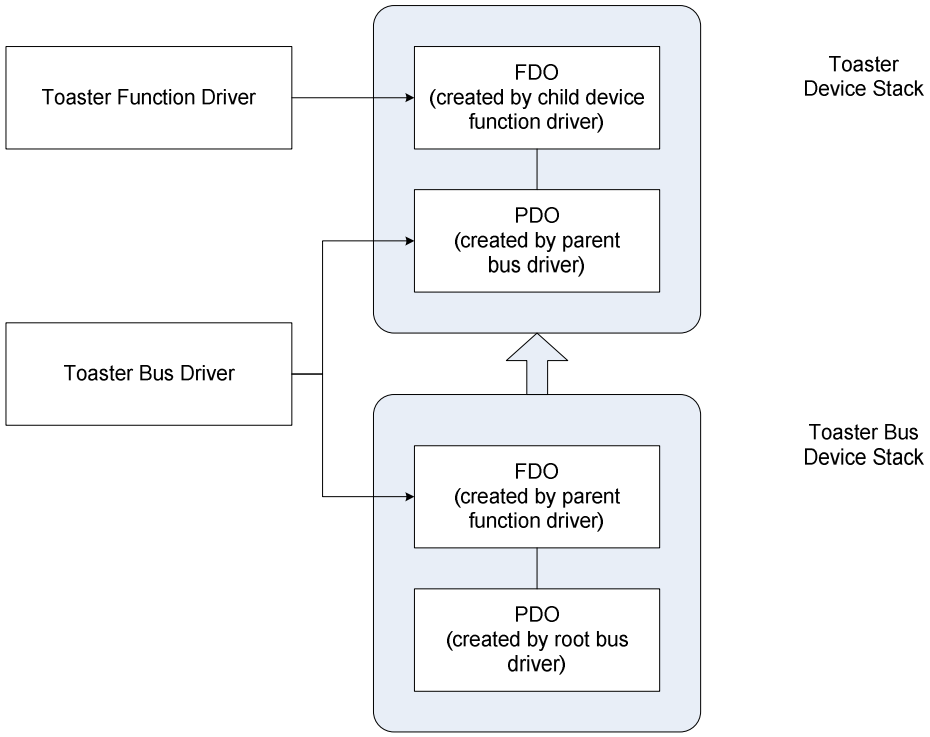

Figure 1. Drivers and device objects in the Toaster device stacks

As Figure 1 shows, the Toaster function driver creates an FDO for the Toaster device. The Toaster bus driver acts as a bus driver for the Toaster device and as a function driver for the Toaster bus. It therefore creates a PDO for the Toaster device and an FDO for the Toaster bus. Because the Toaster bus is a root-enumerated software bus, the root bus driver creates its PDO.

Device Objects and Device Stacks for a Raw PDO

A raw PDO combines the functionality of a PDO and an FDO into a single device object, so that a separate FDO is not required. By creating a raw PDO, a bus driver can either directly manage the device and perform the tasks of a function driver or can provide a separate device stack for “sideband” communication with the device.

KMDF supports such drivers by providing the **WdfPdoInitAssignRawDevice** method, which indicates that the device belongs to a particular device setup class and does not require a function driver. The driver that creates the raw PDO must be able to start and stop the device and handle all read, write, and other I/O requests for the device.

Raw PDOs are useful in two situations:

- Sideband communication.
- Driving a device without a function driver.

Raw PDO for Sideband Communication

On some systems, a driver that is layered above your driver in the device stack might prevent other components from communicating with your device. For example, a higher-level driver might process all I/O control requests (IOCTLs) that are sent to the

device and not pass them any further down the device stack. If your driver supports special functionality by defining one or more custom IOCTLS, your driver must support “sideband” communication by creating a raw PDO so that it can receive those IOCTLs. An application can open a handle to your driver’s raw PDO and send the custom IOCTLs directly to it, therefore bypassing the higher-level driver in the device stack. A bus driver that creates a raw PDO for sideband communication can also be a filter driver or a function driver for that child device.

Figure 2 shows how a raw PDO for sideband communication coexists with the Plug and Play device stack for a device.

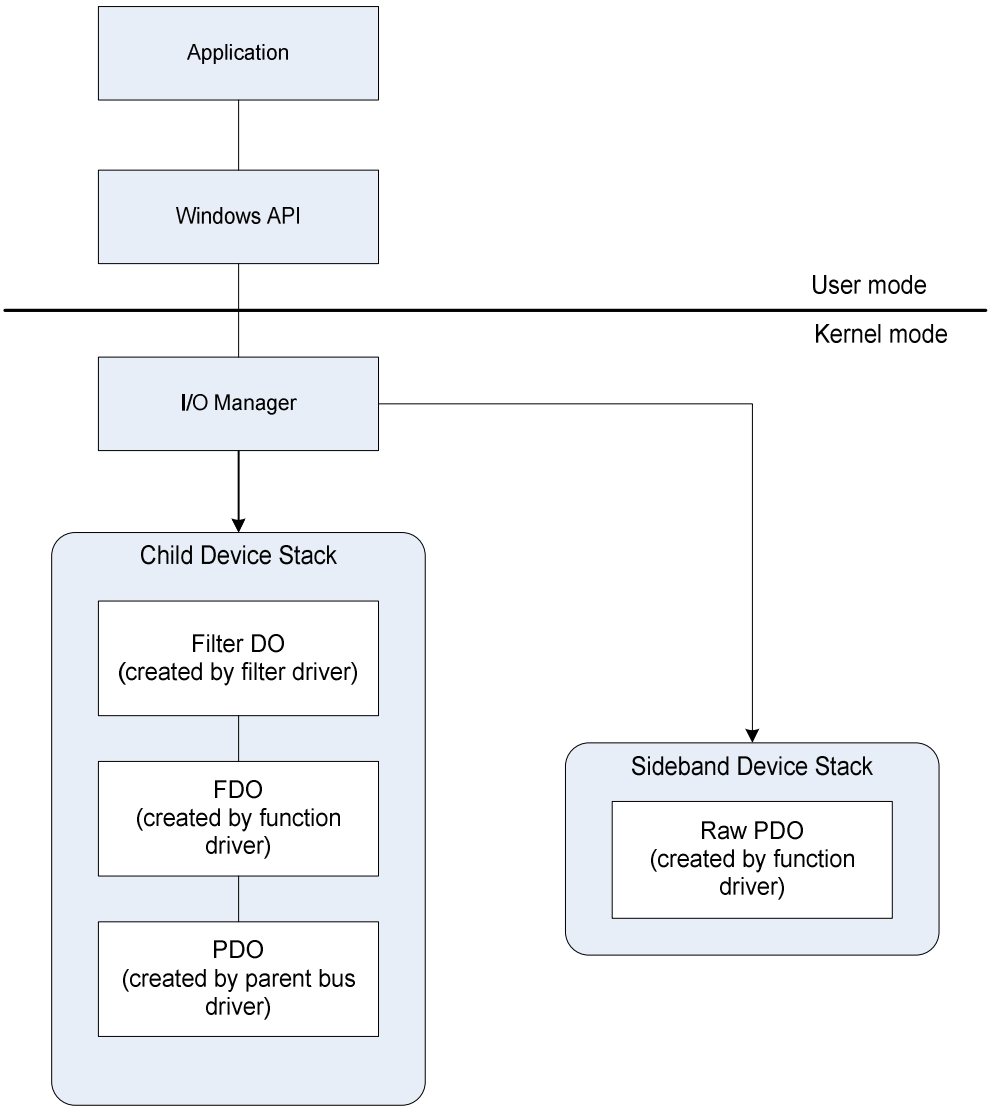

Figure 2. Raw PDO for sideband communication

In Figure 2, the function driver has created a raw PDO for sideband communication to ensure that it receives custom IOCTLs and that such IOCTLs are not filtered out by the higher-level filter driver. An application can open this PDO and send particular I/O requests directly to it, while bypassing the overall Plug and Play device stack for the device. The KbFiltr sample driver shows how to create such a raw PDO for sideband

communication. For details about how to implement a bus driver that has a raw PDO, see “Using a Raw PDO” later in this paper.

Driving a Device without Using a Function Driver

A raw PDO is also appropriate if the bus driver has detailed knowledge about the child device that the PDO represents and therefore can correctly support the features of the child device without a function driver. Figure 3 shows the device stack for a child device that is being driven without a function driver.

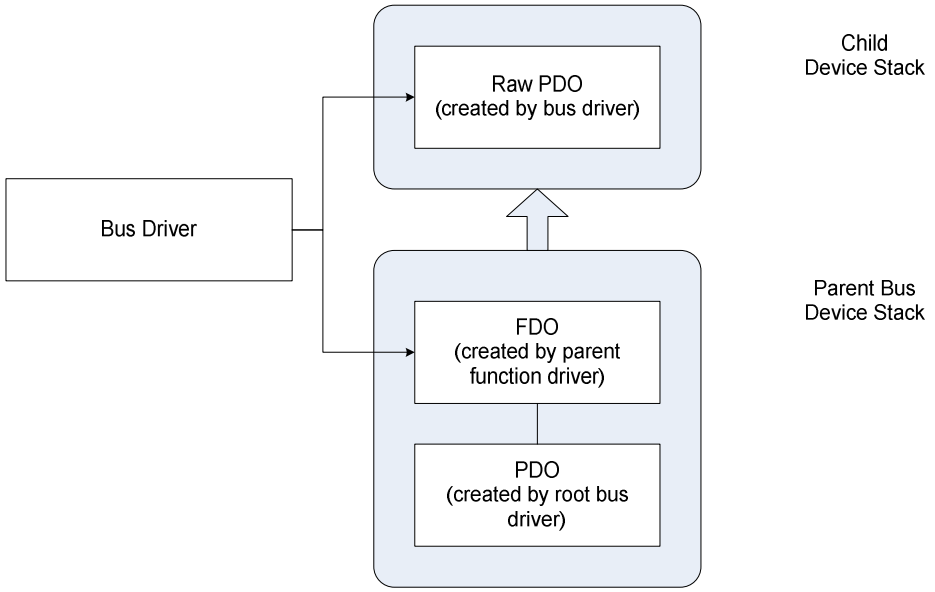

Figure 3. Device stack without a function driver

In Figure 3, the bus driver creates an FDO for the parent bus and creates a raw PDO for the child device. The bus driver performs the tasks of the function driver for the child device in addition to those of the bus driver. The child device stack does not contain a function driver.

If the child device does not have a function driver, the device protocol for the child devices on the bus must provide enough information about the child device so that clients can communicate directly with the bus driver.

To understand the characteristics of such device protocols, consider the PCI bus driver, Pci.sys, and the human interface device (HID) class driver, hidclass.sys.

Pci.sys creates a PDO for each child device that is attached to the PCI bus and supplies information about the resource requirements of each such device. However, PCI.sys does not know anything about the specific features or purpose of any child device, such as a NIC. Therefore, it cannot expose device-specific I/O interfaces for those child devices. Instead, each child device must have a function driver that supports its specific features. Therefore, a raw PDO is unsuitable for PCI.sys.

For many HID devices, however, the protocol itself provides enough information for an application to use a child device without a separate function driver. Microsoft provides in-box function drivers for keyboard, mouse, and joystick devices. The Hidclass.sys driver supports additional HID devices by creating a raw PDO that

represents a high-level collection of HID functions. Because the HID class driver knows how each HID device works, it creates a raw PDO that user-mode applications can open. The top-level HID protocol is rich enough that an application can open the raw PDO for the device by using HID support routines. In this situation, no function driver is required.

If you intend to drive a child device without a function driver by using a raw PDO, the bus driver that creates the raw PDO must be able to start and stop the child device and must handle read, write, and other I/O requests for the child device. By default, the bus driver is the power policy owner (PPO) for the raw PDO.

When a driver creates a raw PDO, the Plug and Play (PnP) manager starts the child device stack immediately because the PDO does not require an FDO to be loaded above it. However, if an INF exists for the child device, the PnP manager starts the child device stack two times: once when it creates the raw PDO and again when it processes the INF. If the PDO already exists when the PnP manager finds the INF, the PnP manager issues requests to query-remove, remove, and start the PDO, just as it does when a user disables and then reenables the device in Device Manager. The framework retains the PDO; the driver is not called again to create a new PDO.

To prevent errors in this situation, a service or application that uses a raw PDO should register for a device interface change notification for GUID_DEVICE_INTERFACE_ARRIVAL and GUID_DEVICE_INTERFACE_REMOVAL. When the removal notification arrives, the application should close its handle to the PDO. When the arrival notification arrives, the application should reopen the handle. The Toastmon sample in %WDKROOT%\Src\Kmdf\Toaster\Toastmon shows how to handle such notification.

KMDF Support for Bus Drivers

KMDF provides the following to support bus driver development:

- PDO-specific methods, types, objects, and events.
- Optional callbacks for PDO-specific Plug and Play requests.
- Optional callbacks for bus-level power-management requests.
- Models for both static and dynamic child-device enumeration.
- Objects, methods, and events to manage collections of child devices.
- Support for “raw PDOs.”

KMDF encapsulates much of the interaction with the Windows kernel and provides default processing for I/O, Plug and Play, and power-management requests, which simplifies driver development. A KMDF bus driver provides callbacks functions only for the requests for which the framework’s defaults are unsuitable. In addition, KMDF supports a hierarchical object model, so that bus drivers can rely on the framework to manage the lifetimes and cleanup of related objects.

Requirements for a KMDF Bus Driver

A typical KMDF bus driver is the function driver for the parent bus and the bus driver for the child devices that are attached to the parent bus. Such a driver is required to:

- Create an FDO for the parent bus, implement function driver features for the parent bus, and report information about the parent bus for use by the PnP manager and the child devices.
- Enumerate child devices on the parent bus and manage collections of child devices.
- Create a PDO for each child device on the parent bus.
- Handle Plug and Play and power-management requests that are directed to the PDOs for the child devices.
- Handle other I/O requests for the PDOs for the child devices.

Sample KMDF Bus Drivers

The WDK includes several sample drivers that demonstrate features of a bus driver. Table 1 lists these drivers and their paths in the %WDKROOT%\Src sample directory.

Table 1. Sample Bus Drivers

Driver name	WDK location	Features
Dynamic Toaster	Kmdf\Toaster\Bus\Dynamic	Dynamic child enumeration
EnumSwitches	Kmdf\Osrusbfx2\Sys\Enumswitches	Raw PDO and dynamic child enumeration
KbFiltr	Kmdf\Kbfiltr\Sys	Raw PDO and static child enumeration
Static Toaster	Kmdf\Toaster\Bus\Static	Static child enumeration

Wherever possible, this paper uses code from these samples.

Parent Bus FDO

Most bus drivers create an FDO for the parent bus and support the usual features that are required in a function driver to handle I/O requests, Plug and Play requests, and power-management requests that are sent to the FDO.

In addition to the usual function driver features, a bus driver can also supply information about its bus for use by drivers in its child device stacks. This information includes the type and number of the bus and the interface that the bus uses to communicate with the child devices. The bus driver supplies the information by filling in a PNP_BUS_INFORMATION structure and calling **WdfDeviceSetBusInformationForChildren** to pass the structure to the framework, typically in the *EvtDriverDeviceAdd* callback function. Table 2 lists the fields in the PNP_BUS_INFORMATION structure.

Table 2. Bus Information in the PNP_BUS_INFORMATION Structure

Field name	Description	Default value
BusTypeGuid	Globally unique identifier (GUID) for the bus, as defined in Wdmguid.h.	-1
LegacyBusType	An enumeration value of INTERFACE_TYPE that specifies the interface that is used to communicate with the child devices. The INTERFACE_TYPE enumeration is defined in the Wdm.h header file.	**InterfaceTypeUndefined**
BusNumber	A ULONG value that uniquely identifies the bus among other buses of the same type on the system.	0xFFFFFFF0

The framework uses this information to respond to IRP_MN_QUERY_BUS_INFORMATION requests from the PnP manager. If the bus driver does not supply the information, the PnP manager uses the default values that are shown in Table 2.

The following sample from the Dynamic Toaster bus driver shows how to fill in this structure:

```
    PNP_BUS_INFORMATION        busInfo;
   //Code omitted
    busInfo.BusTypeGuid = GUID_DEVCLASS_TOASTER;
    busInfo.LegacyBusType = PNPBus; //INTERFACE_TYPE
    busInfo.BusNumber = 0;

    WdfDeviceSetBusInformationForChildren(device, &busInfo);
```

In the example, the Dynamic Toaster bus driver supplies a GUID for the bus type. Drivers for virtual buses should always use GUIDs as bus types. To create a GUID, use the Guidgen or Uuidgen tool. Guidgen is included with the Windows Server® 2003 SP1 WDK and with Visual Studio® 2003 and later versions. Uuidgen is provided with the Platform Software Development Kit (SDK). Do not merely invent a value for a GUID.

Enumeration Models

The bus driver is responsible for enumerating the child devices that are attached to the parent bus. Enumeration involves notifying the PnP manager when child devices are added to the bus or removed from the bus, maintaining and reporting information about each child device, and creating a PDO for each child device.

Depending on the design of the parent bus and the configuration of the system, a bus driver can support static or dynamic enumeration:

- Static enumeration is appropriate if the number, type, and status of child devices that are connected to the bus rarely or never change after system startup.

  A bus driver that supports static enumeration typically creates the child device objects and reports children to the framework during initialization of the parent bus. Such a driver reports changes to child devices only on rare occasions, such as a hardware failure of a child device.

- Dynamic enumeration is appropriate if the number and type of child devices that are connected to the bus can change during typical operation of the system. Dynamic enumeration can occur at any time while the system is running.

  A bus driver that performs dynamic enumeration typically enumerates child devices at startup and includes code to update the child lists if changes occur. The framework and the bus driver manage the children by using one or more WDFCHILDLIST objects, which contain information about the child devices.

Static enumeration and dynamic enumeration are not mutually exclusive, although it is rare to support both enumeration models in the same bus driver. A bus driver that supports dynamic enumeration might use static enumeration for sideband objects.

Static Enumeration

If the child devices that are attached to the parent bus rarely or never change, the bus driver can perform static enumeration. Static enumeration typically occurs in the bus driver's *EvtDriverDeviceAdd* or *EvtDevicePrepareHardware* callback function.

Bus drivers that perform static enumeration are responsible for the following tasks:

- Determining the type and number of child devices.
- Creating a PDO for each child device and initializing information about the child devices.
- Reporting the PDO for each child device to the framework.
- Handling child device removal.
- Handling errors if a child device becomes unresponsive or inaccessible.

The rest of this section describes how to perform static enumeration. It uses the Static Toaster bus driver in %WDKROOT%\Src\Kmdf\Toaster\Bus\Static as an example.

Type and Number of Child Devices

The static enumeration model relies on fixed knowledge of the number and type of child devices that are attached to the parent bus. Typically, this information is either "known" by the bus driver, reported to the bus driver by an application, or stored in the registry. If the information is stored in the registry, it should be located in the devnode for the parent device.

The Static Toaster bus driver reads a value from the registry to determine how many Toaster devices are present. If the value is not present, the bus driver uses a hard-coded default. The bus driver also uses hard-coded serial numbers and hardware IDs for each child device. A bus driver for a physical child device would typically query the child device itself for this information.

Creating a PDO for a Child Device

A bus driver must create a PDO for each of its child devices. To create a PDO, the bus driver:

- Supplies information about the child device in a device initialization structure.
- Initializes an attributes structure and context area for the PDO.

- Calls KMDF to create the PDO.
- Sets Plug and Play and power-management properties in the child-device PDO.

The bus driver can then report the child PDO to the framework.

A bus driver that performs static enumeration creates PDOs for its child devices only at startup, typically in the *EvtDriverDeviceAdd* or *EvtDevicePrepareHardware* callback function. If enumeration does not depend on the state of the child devices, the bus driver can enumerate the children in *EvtDriverDeviceAdd* before the child is powered up. If the child must be in the working state (D0) for the bus driver to enumerate it, the bus driver can perform the enumeration in *EvtDevicePrepareHardware*.

Device Initialization Structure

Before a bus driver can create a PDO, it must provide information about the PDO in a device initialization structure. KMDF provides methods to create the structure and fill in the required information.

To create and fill in a device initialization structure

1. Call **WdfPdoInitAllocate** and pass the handle to the parent bus FDO, which is the parent object for the child device objects. **WdfPdoInitAllocate** returns a handle to a WDFDEVICE_INIT structure.
2. Set the child device type by calling the **WdfDeviceInitSetDeviceType** method.
3. Call additional **WdfDeviceInitXxx** methods as appropriate to supply information about the driver and device.

   Bus drivers that create raw PDOs typically call **WdfDeviceInitSetExclusive** to ensure that only one handle can be open to the PDO at a time. Some bus drivers also call **WdfDeviceInitSetCharacteristics** to name the device object. By default, the PnP manager generates a unique name for the PDO if the driver does not name the PDO. As a security measure, drivers should use the default name that the PnP manager generates unless some other name is absolutely required.
4. Call other **WdfPdoInitXxx** methods to fill in additional information about the child device.

   The **WdfPdoInitXxx** methods supply information that the framework uses to respond to PnP manager queries about the child device. which can start to arrive immediately after the framework creates the underlying WDM PDO. Therefore, the bus driver must supply the information about the child device before it creates the PDO.

Table 3 lists the PDO-specific information that the bus driver can supply in the WDFDEVICE_INIT structure.

Table 3. Information in the PDO Initialization Structure

Information	Initialization method	Comments
Compatible ID	**WdfPdoInitAddCompatibleID**	A vendor-defined string that contains an ID for the child device that does not exactly match the child device but is nevertheless compatible with it. The compatible ID is less specific than the hardware ID. A device can have more than one compatible ID, and the bus driver must add them in order from the most suitable to the least suitable. Setup uses the compatible IDs to match the child device to the INF file to be used to install the function driver for the child device.
Default locale	**WdfPdoInitSetDefaultLocale**	Locale identifier (LCID). For details, see "Locale Identifier Constants and Strings" on MSDN®.
Device ID	**WdfPdoInitAssignDeviceID**	Required. A vendor-defined string that contains the most specific ID for the child device. A device has only one device ID. Setup uses this ID to match the child device to an INF file. The bus driver must call **WdfPdoAssignDeviceId** for a child device before it creates the PDO for the child device.
Device text	**WdfPdoInitAddDeviceText**	A locale-specific string that describes the child device. The PnP manager displays this text briefly while it searches for a matching INF file.
Hardware ID	**WdfPdoInitAddHardwareID**	Required only on Windows 2000. A vendor-defined string that contains an ID for the child device. A device can have more than one hardware ID, and the bus driver must add them in order from the most suitable to the least suitable. The hardware ID is more specific than the compatible ID. Setup uses these IDs to match the child device to an INF file. The first hardware ID that is added should be the device ID.
Instance ID	**WdfPdoInitAssignInstanceID**	Required. A device identification string that distinguishes a child device from other devices of the same type on a machine.
Pointers to callback functions	**WdfPdoInitSetEventCallbacks**	Pointers to the event callback functions that the bus driver implements for the PDO.
Raw PDO	**WdfPdoInitAssignRawDevice**	An indication that the bus driver supports the device in raw mode.

A bus driver is not required to supply all the types of IDs. In Windows XP and later versions, only a device ID and an instance ID are required. Windows 2000 requires a device ID, an instance ID, and a hardware ID.

PDO Initialization Example

The Toaster bus drivers initialize the following information for the child PDOs:

- Device type
- Device ID
- Hardware ID
- Compatible ID
- Instance ID
- Device text
- Default locale

The following code sample shows the initialization code, which is the same in both the Static and Dynamic Toaster bus drivers:

```
DECLARE_CONST_UNICODE_STRING(compatId, BUSENUM_COMPATIBLE_IDS);
UNICODE_STRING deviceId;

pDeviceInit = WdfPdoInitAllocate(Device);
if (pDeviceInit == NULL) {
    status = STATUS_INSUFFICIENT_RESOURCES;
    goto Cleanup;
}

WdfDeviceInitSetDeviceType(pDeviceInit, FILE_DEVICE_BUS_EXTENDER);

// HardwareID is passed to the Toaster driver by the Toaster
// application.

RtlInitUnicodeString(&deviceId,HardwareIds);
status = WdfPdoInitAssignDeviceID(pDeviceInit, &deviceId);
if (!NT_SUCCESS(status)) {
    goto Cleanup;
}
status = WdfPdoInitAddHardwareID(pDeviceInit, &deviceId);
if (!NT_SUCCESS(status)) {
    goto Cleanup;
}
status = WdfPdoInitAddCompatibleID(pDeviceInit, &compatId);
if (!NT_SUCCESS(status)) {
    goto Cleanup;
}
status =  RtlUnicodeStringPrintf(&buffer, L"%02d", SerialNo);
if (!NT_SUCCESS(status)) {
    goto Cleanup;
}
status = WdfPdoInitAssignInstanceID(pDeviceInit, &buffer);
if (!NT_SUCCESS(status)) {
    goto Cleanup;
}
status = RtlUnicodeStringPrintf(&buffer,
                                L"Microsoft_Eliyas_Toaster_%02d",
                                SerialNo );
if (!NT_SUCCESS(status)) {
    goto Cleanup;
}
status = WdfPdoInitAddDeviceText(pDeviceInit,
                                 &buffer,
                                 &deviceLocation,
                                 0x409);
```

```
if (!NT_SUCCESS(status)) {
    goto Cleanup;
}
WdfPdoInitSetDefaultLocale(pDeviceInit, 0x409);
```

If the bus driver supports optional event callback functions for the PDO, the bus driver next registers them with the framework by initializing a WDF_PDO_EVENT_CALLBACKS structure and then calling **WdfPdoInitSetEventCallbacks**. For more information on these callbacks, see "Optional PDO Event Callback Functions" later in this paper.

Attributes Structure and Context Area

After the bus driver has initialized the settings and the optional event callbacks, it initializes an attributes structure and a device context area for the PDO by calling WDF_OBJECT_ATTRIBUTES_INIT or WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE. These are the same functions that a function or filter driver uses to initialize an attributes structure and a device context area for an FDO or filter DO. For example:

```
WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&pdoAttributes,
                                        PDO_DEVICE_DATA);
```

The example uses WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE to initialize the attributes structure and to set PDO_DEVICE_DATA as the type of the device context structure.

PDO Creation

The bus driver can now create the PDO by calling **WdfDeviceCreate**. It passes the device initialization structure and the attributes structure as parameters and receives a handle to the PDO, as in the following example:

```
status = WdfDeviceCreate(&pDeviceInit, &pdoAttributes, &hChild);
if (!NT_SUCCESS(status)) {
    goto Cleanup;
}
```

After successful creation of the PDO, KMDF frees the memory that was allocated to the WDFDEVICE_INIT structure and sets its handle to NULL. The bus driver must not call any **WdfPdoInitXxx** or **WdfDeviceInitXxx** methods for the PDO after successfully returning from **WdfDeviceCreate**.

Device Plug and Play and Power-Management Capabilities

After the bus driver creates the PDO, it must supply additional information about the Plug and Play and power-management features of the child device hardware. The framework uses this information to respond to queries from the PnP manager. These settings are propagated through the child's device stack and are used to initialize values for the function driver and filter drivers for the child device.

To report the Plug and Play features of the bus and the device, the bus driver initializes and fills in a WDF_DEVICE_PNP_CAPABILITIES structure. Table 4 lists the relevant members of this structure. A bus driver is not required to supply values for all members; it should fill in only the members that apply to the child device.

Table 4. Members of the Plug and Play Capabilities Structure

Field name	Default value	Description
Address	0xFFFFFFFF (-1)	Address of the child device on the bus.
DockDevice	**WdfFalse**	Whether the child device is a docking station.
EjectSupported	**WdfFalse**	Whether the child device can be ejected from its slot on the bus.

Field name	Default value	Description
HardwareDisabled	**WdfFalse**	Whether the child device is disabled.
LockSupported	**WdfFalse**	Whether the child device can be locked in its slot to prevent ejection.
NoDisplayInUI	**WdfFalse**	Whether the child device should not be displayed in Device Manager.
Removable	**WdfFalse**	Whether the child device can be removed while the system is running. If this value is **WdfTrue** and if **SurpriseRemovalOK** is **WdfFalse**, the bus driver supports device removal only through the Unplug or Eject Hardware application.
SilentInstall	**WdfTrue** for raw PDOs; otherwise, **WdfFalse**	Whether installation of the child device should occur without the display of any user interface items (except error dialog boxes), if the device is already preinstalled in the driver store.
SurpriseRemovalOK	**WdfFalse**	Whether a user can remove the child device without using the Unplug or Eject Hardware program. If **SurpriseRemoveOk** is **WdfTrue**, **Removable** must also be **WdfTrue**.
UINumber	-1	User-perceived slot number that can help a user identify the physical location of the child device.
UniqueID	**WdfFalse**	Whether the child device's instance ID is unique across the system.

The bus driver reports power capabilities by initializing and filling a WDF_DEVICE_POWER_CAPABILITIES structure. Table 5 lists the members of this structure. The bus driver fills in the members to indicate the power-management features of the bus and the child device. As with the Plug and Play capabilities, the framework propagates this information through the child's device stack and uses it as defaults for higher-level function and filter drivers.

Table 5. Members of the Power Capabilities Structure

Member name	Default value	Description
D1Latency	-1	Approximate time, in 100-nanosecond units, that the child device requires to return to the D0 state from D1.
D2Latency	-1	Approximate time, in 100-nanosecond units, that the child device requires to return to the D0 state from D2.
D3Latency	-1	Approximate time, in 100-nanosecond units, that the child device requires to return to the D0 state from D3.
DeviceD1	**WdfUseDefault**	Whether the child device supports D1.
DeviceD2	**WdfUseDefault**	Whether the device supports D2.
DeviceState [PowerSystemMaximum]	**PowerDeviceMaximum**	The most-powered D*x* state that the child device can support for each system power state.
DeviceWake	**PowerDeviceMaximum**	The lowest-powered D*x* state from which the child device can send a wake signal to the system.

Member name	Default value	Description
IdealDxStateForSx	**PowerDeviceMaximum**	The D*x* state that the child device should enter when it is not enabled to wake the system and the system enters an S*x* state.
SystemWake	**PowerSystemMaximum**	The lowest-powered S*x* state from which the child device can send a wake signal to the system.
WakeFromD0	**WdfUseDefault**	Whether the child device can respond to a wake signal while in D0.
WakeFromD1	**WdfUseDefault**	Whether the child device can respond to a wake signal while in D1.
WakeFromD2	**WdfUseDefault**	Whether the child device can respond to a wake signal while in D2.
WakeFromD3	**WdfUseDefault**	Whether the child device can respond to a wake signal while in D3.

The **WdfUseDefault** enumeration value indicates that the framework uses whatever value was supplied by a lower driver in the device stack. For example, if a bus driver specifies **WdfTrue** for **DeviceD1** and the function driver specifies **WdfUseDefault**, the framework uses **WdfTrue**. Similarly, **PowerDeviceMaximum** and **PowerSystemMaximum** indicate that the framework should use whatever value the system already has stored for that member.

To initialize the Plug and Play and power capabilities structures, the bus driver calls WDF_DEVICE_PNP_CAPABILITIES_INIT and WDF_DEVICE_POWER_CAPABILITIES_INIT, respectively. Then it fills in the relevant members of the structure for the child device and calls the **WdfDeviceSetPnpCapabilities** and **WdfDeviceSetPowerCapabilities** methods to pass the information to the framework.

The following example shows how the Toaster bus drivers report the Plug and Play and power-management capabilities for a child device:

```
    WDF_DEVICE_PNP_CAPABILITIES_INIT(&pnpCaps);
    pnpCaps.Removable         = WdfTrue;
    pnpCaps.EjectSupported    = WdfTrue;
    pnpCaps.SurpriseRemovalOK = WdfTrue;
    pnpCaps.Address  = SerialNo;
    pnpCaps.UINumber = SerialNo;
    WdfDeviceSetPnpCapabilities(hChild, &pnpCaps);

    WDF_DEVICE_POWER_CAPABILITIES_INIT(&powerCaps);
    powerCaps.DeviceD1 = WdfTrue;
    powerCaps.WakeFromD1 = WdfTrue;
    powerCaps.DeviceWake = PowerDeviceD1;
    powerCaps.DeviceState[PowerSystemWorking]   = PowerDeviceD0;
    powerCaps.DeviceState[PowerSystemSleeping1] = PowerDeviceD1;
    powerCaps.DeviceState[PowerSystemSleeping2] = PowerDeviceD3;
    powerCaps.DeviceState[PowerSystemSleeping3] = PowerDeviceD3;
    powerCaps.DeviceState[PowerSystemHibernate] = PowerDeviceD3;
    powerCaps.DeviceState[PowerSystemShutdown] = PowerDeviceD3;
    WdfDeviceSetPowerCapabilities(hChild, &powerCaps);
```

Reporting Static Children to the Framework

After the bus driver creates the PDO for a child device, it must notify the framework about the child by calling **WdfFdoAddStaticChild** and passing a handle to the parent bus FDO and a handle to the newly created child-device PDO. For example:

```
status = WdfFdoAddStaticChild(Device, hChild);
    if (!NT_SUCCESS(status)) {
        goto Cleanup;
    }
```

The framework maintains information about the static child devices on behalf of the bus driver. Internally, the framework keeps track of the child devices by their WDF device objects.

Iterating through Static Children

A bus driver might search for a particular child device so that it can verify data about the child or get a handle to the child's device object to mark it as missing. KMDF provides the following methods for iterating through the statically enumerated child devices:

- **WdfFdoLockStaticChildListForIteration**
- **WdfFdoRetrieveNextStaticChild**
- **WdfFdoUnlockStaticChildListFromIteration**

  Note: Although the names of the lock and unlock methods imply otherwise, the framework does not expose a static child list to drivers in the same way in which it exposes a dynamic child list. Instead, the framework maintains an internal list. A driver can manipulate static children only by accessing their individual WDFDEVICE objects.

Before a bus driver scans through the child devices, it must call **WdfFdoLockStaticChildListForIteration** to acquire a lock for reading through the children. If another thread tries to update a child device while the bus driver holds the lock, KMDF stores the changes and applies them after the bus driver releases the lock. To release the lock when the scan is complete, the bus driver calls **WdfFdoUnlockStaticChildListFromIteration**.

After a bus driver acquires the lock, it calls **WdfFdoRetrieveNextStaticChild** multiple times to retrieve a handle to the PDO for each child device. The first call to this method returns the first child-device PDO in the list. Later calls by the bus driver to the same function return additional child-device PDOs. After the bus driver has retrieved the last child-device PDO, the function returns NULL.

WdfFdoRetrieveNextStaticChild takes the following three parameters:

- The handle to the parent bus FDO.
- A WDFDEVICE variable into which the framework returns a handle to the returned child-device PDO.
- A WDF_RETRIEVE_CHILD_FLAGS enumeration value that indicates which child devices to return.

Table 6 lists the values of the WDF_RETRIEVE_CHILD_FLAGS enumeration.

Table 6. Values of the WDF_RETRIEVE_CHILD_FLAGS Enumeration

Enumeration value	Description
WdfRetrieveAddedChildren	Returns all present and pending child devices.
WdfRetrieveAllChildren	Returns present, pending, and missing child devices.
WdfRetrieveMissingChildren	Returns child devices that the bus driver has previously marked as missing by calling **WdfPdoMarkMissing**.
WdfRetrievePendingChildren	Returns child devices that the bus driver has reported as present, but which the framework has not yet reported to the PnP manager.
WdfRetrievePresentChildren	Returns all child devices for which the bus driver has already created a device object and that the framework has already reported to the PnP manager.

Locks for Statically Enumerated Children

Both the framework and the bus driver must use locks to prevent conflicts during the addition or removal of static child devices or the deletion of their PDOs. KMDF implements a lock that lets one thread have write access or multiple threads to have read access to all the statically enumerated children. The framework and the bus driver use this lock as follows:

- KMDF automatically locks the child list when a driver calls **WdfFdoAddStaticChild**. The driver is not required to call a KMDF locking method or to supply its own lock simply to add child devices to the list.
- The **WdfFdoLockStaticChildListForIteration** method acquires the lock for read access, as described in the previous section. If another thread tries to add or remove a child device while the lock is held, KMDF stores the changes and applies them after the bus driver unlocks the list.

The framework's lock protects the integrity of the framework's internal data structure by allowing one writer or many readers at any one time. However, this lock does not prevent conflicts in the data within the child-device PDOs.

If your bus driver must ensure that child device data is unique—for example, if each child device must have a unique serial number—the bus driver must create an additional lock that restricts read access to a single thread at any time. While the bus driver holds the lock, it can iterate through the children, ensure that the data for the new child is unique, create the child's PDO, and report the child PDO to the framework. Without such a lock, a second thread could simultaneously iterate through the child devices and try to add a child device that has the same data. The bus driver must use a passive-level locking mechanism—a wait lock—because it calls **WdfDeviceCreate** while it holds the lock and **WdfDeviceCreate** must be called at PASSIVE_LEVEL.

The Static Toaster bus driver sample shows how to use such a lock. In the *EvtDriverDeviceAdd* callback function, the sample creates a wait lock to protect the child device data, as follows:

```
deviceData = FdoGetData(device);
WDF_OBJECT_ATTRIBUTES_INIT(&attributes);
attributes.ParentObject = device;
status = WdfWaitLockCreate(&attributes, &deviceData->ChildLock);
    if (!NT_SUCCESS(status)) {
        return status;
    }
```

In the example, FdoGetData is the accessor function for the device context area of the parent bus FDO. The bus driver gets a pointer to the context area, initializes an attributes structure, and then sets the parent bus FDO as the parent object of the lock so that the lifetime of the lock object matches that of the FDO. It stores the handle to the lock in the ChildLock field of the device context area. The bus driver uses the lock when it scans the child devices during device enumeration.

Static Iteration Example

In the Static Toaster bus sample, the Bus_PlugInDevice function iterates through the child devices and compares the serial number of each existing child device to that of a newly plugged-in child device to ensure that the serial number of the new child device is unique. The bus driver stores the serial number in the context area of the child-device PDO. The following is the code for this function:

```
NTSTATUS
Bus_PlugInDevice(
    __in WDFDEVICE  Device,
    __in PWCHAR     HardwareIds,
    __in ULONG      SerialNo
    )

{
    NTSTATUS         status = STATUS_SUCCESS;
    BOOLEAN          unique = TRUE;
    WDFDEVICE        hChild;
    PPDO_DEVICE_DATA pdoData;
    PFDO_DEVICE_DATA deviceData;

    PAGED_CODE ();
/* Get pointer to FDO context area.
    deviceData = FdoGetData(Device);
    hChild = NULL;
/* Acquire locks.
    WdfWaitLockAcquire(deviceData->ChildLock, NULL);
    WdfFdoLockStaticChildListForIteration(Device);
/* Get child PDO.
    while ((hChild = WdfFdoRetrieveNextStaticChild(Device,
                      hChild, WdfRetrieveAddedChildren)) != NULL) {
        pdoData = PdoGetData(hChild);
        /* Compare serial numbers.
        if (SerialNo == pdoData->SerialNo) {
            unique = FALSE;
            status = STATUS_INVALID_PARAMETER;
            break;
        }
    }
```

```
/* Create new PDO if number is unique.
    if (unique) {
        status = Bus_CreatePdo(Device, HardwareIds, SerialNo);
    }
/* Release locks in reverse order of acquisition.
    WdfFdoUnlockStaticChildListFromIteration(Device);
    WdfWaitLockRelease(deviceData->ChildLock);

    return status;
}
```

In the sample, FdoGetData and PdoGetData are the accessor functions for the FDO and PDO context areas, respectively.

The Bus_PlugInDevice function acquires both the bus driver–created wait lock and the framework's lock before its first call to **WdfFdoRetrieveNextStaticChild**. The bus driver requests all the present and added child devices, so that it can check the serial numbers of any children that are pending addition as well as those that are already known to the framework. If the serial number of the newly plugged-in child device is unique, the bus driver calls the internal Bus_CreatePdo function, which creates a PDO for the child device and calls **WdfFdoAddStaticChild** to add the child.

Child Device Removal in the Static Enumeration Model

Even if a child device is permanently physically attached to the bus, the bus driver must include code to handle device removal. A child device can be removed in several ways:

- The user stops the child device by using the Safely Remove Hardware application.
- The user disables the child device by using Device Manager.
- The user uninstalls the child device by using Device Manager.
- The bus to which the child device is attached is disabled or removed.
- The user removes the child device by surprise, without using Device Manager or the Safely Remove Hardware application.

Removal takes two forms: graceful removal (also known as "orderly removal") and surprise removal.

Graceful Removal

Graceful removal occurs when the user notifies the system before removing or disabling the child device, by using either Device Manager or the Safely Remove Hardware application.

If the child device remains physically present—that is, if the user does not uninstall or eject the child device—the PnP manager retains the child's PDO but marks it as "not started." No special processing or callback functions are required in the bus driver; graceful removal is accomplished through the standard Plug and Play and power-management callbacks.

If the user later reenables the child device from Device Manager, the framework uses the retained child PDO and reenables the child device. To do this, the framework follows the same sequence of callbacks as at startup, beginning with the bus driver's *EvtDevicePrepareHardware* callback function.

If the user physically removes a device, either the bus receives an interrupt or the device fails to respond to the bus in a particular, device-specific manner. At this point, the bus driver determines that the device has been removed and reports the removal to the framework by calling **WdfPdoMarkMissing**. This method informs the framework that the child device is no longer accessible. The framework deletes the child's PDO. The bus driver must not delete the PDO.

Surprise Removal

Surprise removal occurs when the user physically removes the child device from its port or slot without using Device Manager or the Safely Remove Hardware application. To restart a child device that has been physically removed and reinserted, the PnP manager must reenumerate the child device and the bus driver must create a new PDO for the child device.

A bus driver that performs static enumeration handles surprise removal by calling **WdfPdoMarkMissing**. KMDF flags the child's PDO as representing a device that is not present, but does not delete the PDO.

In the Static Toaster bus driver, an application initiates surprise removal by sending an IOCTL that contains the serial number of the toaster that was unplugged from the bus. In response, the bus driver searches through the child list to find the child device that corresponds to the serial number and then calls **WdfPdoMarkMissing** to mark the child device as missing. The application can also indicate that all the Toasters were unplugged from the Toaster bus. In this case, the bus driver simply marks all the child devices as missing.

The following example shows how the Static Toaster bus driver handles surprise removal:

```
NTSTATUS
Bus_UnPlugDevice(
    WDFDEVICE   Device,
    ULONG       SerialNo
    )
{
    PPDO_DEVICE_DATA pdoData;
    BOOLEAN          found = FALSE;
    BOOLEAN          plugOutAll;
    WDFDEVICE        hChild;
    NTSTATUS         status = STATUS_INVALID_PARAMETER;

    PAGED_CODE ();

/*  Unplug one device or all devices?
    plugOutAll = (0 == SerialNo) ? TRUE : FALSE;
    hChild = NULL;
/*  Acquire lock.
    WdfFdoLockStaticChildListForIteration(Device);
/*  Get a child PDO.
    while ((hChild = WdfFdoRetrieveNextStaticChild(Device,
                     hChild, WdfRetrieveAddedChildren)) != NULL) {
/*      Unplug all devices.
        if (plugOutAll) {
            status = WdfPdoMarkMissing(hChild);
            if(!NT_SUCCESS(status)) {
                KdPrint(("WdfPdoMarkMissing failed 0x%x\n", status));
                break;
            }
```

```
            found = TRUE;
        }
        else {
/*      Is this the device to unplug?
            pdoData = PdoGetData(hChild);
            if (SerialNo == pdoData->SerialNo) {
/*          Yes, mark it missing.
                status = WdfPdoMarkMissing(hChild);
                if(!NT_SUCCESS(status)) {
                    KdPrint(("WdfPdoMarkMissing failed 0x%x\n",
                            status));
                    break;
                }
                found = TRUE;
                break;
            }
        }
    }
    WdfFdoUnlockStaticChildListFromIteration(Device);
    if (found) {
        status = STATUS_SUCCESS;
    }

    return status;
}
```

The bus driver uses only the framework's lock when it marks the child device as missing. If another thread tries to mark the same child device as missing, one of the attempts fails but the child device is still marked as missing and the integrity of the list remains intact.

The framework deletes the PDO. The bus driver must not delete the PDO.

Dynamic Enumeration

If the number or type of child devices is likely to change while the system is running, the bus driver should support dynamic enumeration. In the dynamic enumeration model, the bus driver and the framework keep track of the child devices by using a child list, which is a WDFCHILDLIST object. The child list identifies child devices by using an identification structure. The bus driver defines the layout and contents of the structure, configures the child list, and updates the child list during bus operation to reflect the addition and removal of child devices. The framework uses the child list to supply information about the child devices to the PnP manager.

The bus driver must maintain an accurate child list so that the framework can provide accurate information to the PnP manager. The PnP manager can request such information at any time—not just at startup. Consequently, you must design your bus driver to update the child list in a timely manner when hardware changes occur.

At minimum, a bus driver that performs dynamic enumeration must handle the following tasks:

- Configure the initial child list for the framework.
- Provide an identification description that uniquely identifies each child device.
- Report each child device to the framework.
- Implement an *EvtChildListCreateDevice* callback function, which creates a PDO for a child device in response to the framework's request.

- Handle the removal of child devices.
- Update the child list in response to changes in child-device availability or status.

This section summarizes the steps in dynamic enumeration and a general discussion of dynamic child lists, followed by discussions of the individual tasks in dynamic enumeration.

Overview of Dynamic Enumeration

Device discovery and enumeration in the dynamic enumeration model occur in the following order:

1. At startup, the system loads the bus driver and the framework calls the bus driver's *EvtDriverDeviceAdd* callback function.
2. The *EvtDriverDeviceAdd* callback function configures a default child list before it creates the FDO for the parent bus. This function should also call **WdfDeviceSetBusinformationForChildren** to supply information about the bus for use by the PnP manager and any child devices.

   When the bus driver calls the framework to create the parent FDO, the framework creates the default child list according to the bus driver's child-list configuration.
3. The bus driver enumerates the child devices on the bus and populates the child list with information about them.

   A bus driver can perform dynamic enumeration any time during or after its *EvtDevicePrepareHardware* callback function runs and before its *EvtDeviceRemoveHardware* callback function runs, depending on the design of the hardware. Typically, drivers perform enumeration in one of the following callback functions:

   - In *EvtDevicePrepareHardware*, if the bus driver can enumerate its children before the bus has entered the D0 state.
   - In *EvtDeviceD0Entry*, which the framework calls after the bus is in D0 but before interrupts are connected.
   - In *EvtChildListScanForChildren*, which the framework calls after *EvtDeviceD0PostInterruptsEnabled*. At this point, the bus is in D0, interrupts are connected, and post-interrupt processing is complete.
4. The framework later calls the bus driver's *EvtChildListCreateDevice* callback function once for each child device to create a PDO for the child device.
5. As children are added to the bus and removed from the bus, the bus driver updates the child list and notifies the framework. Depending on the design of the bus and device hardware, a driver might update the list in any of several places. For example:

   - If the bus generates an interrupt when a child device is added or removed, the bus driver can update the child list from within its *EvtInterruptDpc* callback function.
   - If the bus driver is notified about the addition or removal of child devices through a mechanism other than an interrupt, the bus driver can update the child list from within the code that processes the notification.

- If the function driver for a child device can request reenumeration by sending to its PDO the Plug and Play IRP_MN_QUERY_INTERFACE request for REENUMERATE_SELF_INTERFACE_STANDARD, the bus driver can optionally implement an *EvtChildListDeviceReenumerated* callback function to approve or veto such a request.

Child Lists

The framework creates and manages child lists for bus drivers that perform dynamic enumeration. A child list is a WDFCHILDLIST object that contains identifying information about the child devices on the bus and stores pointers to the bus driver's callback functions that support child device enumeration and identification. The framework creates an empty default child list when a bus driver creates the parent bus FDO.

Most bus drivers use only one child list, although a bus driver can create and use additional child lists by calling **WdfChildListCreate**. By using more than one child list, a bus driver can group children by a common property. As an example, consider a bus driver that can enumerate children over two different protocols, A and B. The bus driver can group the children from each protocol into separate child lists. If one of the protocols is disconnected, the bus driver can mark all that protocol's children missing with a single call.

Contents of the Child List

Because of the wide variation in devices, KMDF defines a driver-configurable child-list object for use in dynamic enumeration. The child-list object contains driver-defined data structures that, in turn, contain information that uniquely identifies the child devices on the bus. Before the framework creates a child list, the bus driver must configure the list by specifying the size and layout of the driver-defined data structures and providing pointers to any driver-defined callback functions that the framework should call to modify and compare those data structures. The framework then creates the child list according to the bus driver's configuration.

A child list can contain two types of structures for each child device:

- An identification description structure, which is required.
- An address description structure, which is optional.

Before a bus driver creates an FDO for the parent bus or dynamically enumerates any child devices, it must set up a child-list configuration structure for the default child list. The driver calls the framework to initialize the structure and supplies information about the identification description structure, the address description structure, and any related callback functions. The bus driver typically performs these tasks in the *EvtDriverDeviceAdd* callback function.

Identification Description Structure

An identification description structure uniquely identifies a child device. When a child device arrives on the bus, KMDF uses the information in the structure to determine whether the child device is a new device or an existing device.

The first member of the identification description structure must be a WDF_CHILD_IDENTIFICATION_DESCRIPTION_HEADER structure, which is defined by KMDF. All other members are driver defined and contain child device–specific data. Typically, an identification description includes the device ID string together with a serial number, bus slot, or similar information that uniquely identifies the child device. The following code example shows the identification description structure for a hypothetical IEEE 1394 device:

```
typedef struct _CHILD_DEVICE_ID_DESCRIPTION {
    WDF_CHILD_IDENTIFICATION_DESCRIPTION_HEADER     IdHeader;
    ULONG                                           DeviceType;
    ULARGE_INTEGER                                  InstanceId;
    ULONG                                           UnitSpecId;
    ULONG                                           UnitSwVersion;
    WCHAR                                  DeviceId[MaxDeviceNameSizeCch];
} CHILD_DEVICE_ID_DESCRIPTION, *PCHILD_DEVICE_ID_DESCRIPTION;
```

The framework has no knowledge of the contents of the identification description structure beyond the header. When the driver configures the child list, it supplies the length of the structure and, optionally, one or more callback functions that the framework invokes to manage the identification descriptions.

By default, the framework calls **RtlCopyMemory** to copy or duplicate an identification description structure and calls **RtlCompareMemory** to compare two identification description structures. However, if the structure contains description-relative pointers or dynamically allocated memory, a bus driver can provide optional callback functions that the framework calls instead of **RtlCopyMemory** or **RtlCompareMemory**. Table 7 lists these optional callback functions.

Table 7. Identification Description Callback Functions

Callback function	Description
EvtChildListIdentificationDescriptionCompare	Compares the contents of two identification description structures.
EvtChildListIdentificationDescriptionCopy	Copies the contents of one identification description structure to another existing identification description structure.
EvtChildListIdentificationDescriptionDuplicate	Creates a new identification description structure by duplicating an existing identification description structure and, if necessary, allocating any additional memory that is required for the new structure.
EvtChildListIdentificationDescriptionCleanup	Deallocates any memory in a duplicated identification description structure that *EvtChildListIdentificationDescriptionDuplicate* allocated.

The *EvtChildListIdentificationDescriptionCleanup* callback function is required only if an identification description structure contains memory that *EvtChildListIdentificationDescriptionDuplicate* allocated. *EvtChildListIdentificationDescriptionCleanup* must deallocate only the memory that *EvtChildListIdentificationDescriptionDuplicate* allocated. It must not delete the identification description structure itself. The framework frees the structure.

Address Description Structure

If the address of the child device on the bus can change while the device is plugged in, the bus driver must supply an address description in addition to an identification description.

For example, the IEEE 1394 bus assigns a dynamic address to each device on the bus and periodically updates and reassigns these addresses. The bus address contains a generation count that indicates the current generation of the bus. The generation count is required in each asynchronous I/O request to a device on the bus. An IEEE 1394 bus driver must therefore include both the current generation count and with the 1394 address for the device in the address description structure. The following code example shows the definition of such a structure:

```
typedef struct _CHILD_DEVICE_ADDRESS_DESCRIPTION {
    WDF_CHILD_ADDRESS_DESCRIPTION_HEADER  AddressHeader;
    ULONG                                 GenerationCount;
    ULONG                                 LocalNodeId;
    ULONG                                 DeviceNodeId;
    UCHAR                                 SpeedToNode;
    ULONG                                 UnitDirectoryIndex;
    ULONG                                 UnitDependentDirectoryIndex;
    ULONG                                 VendorLeafIndex;
    ULONG                                 ModelLeafIndex;
    CONFIG_ROM                            ConfigRom;

}CHILD_DEVICE_ADDRESS_DESCRIPTION, *PCHILD_DEVICE_ADDRESS_DESCRIPTION;
```

The layout of the address description structure follows the same pattern as the identification description structure. The first member of the structure must be a WDF_CHILD_ADDRESS_DESCRIPTION_HEADER structure. All other members are driver defined and contain child device–specific data.

As with the identification description structure, by default the framework calls **RtlCopyMemory** to copy or duplicate an address description structure. However, if the structure contains description-relative pointers or dynamically allocated memory, a bus driver can provide optional callback functions that the framework calls instead of **RtlCopyMemory**. Table 8 lists these optional callback functions.

Table 8. Address Description Callback Functions

Callback function	Description
EvtChildListAddressDescriptionCopy	Copies the contents of one address description structure to another existing address description structure.
EvtChildListAddressDescriptionDuplicate	Creates a new address description structure by duplicating an existing address description structure and, if necessary, allocating any additional memory that is required for the new structure.
EvtChildListAddressDescriptionCleanup	Deallocates any memory in a duplicated address description structure that *EvtChildListAddressDescriptionDuplicate* allocated.

The *EvtChildListAddressDescriptionCleanup* callback function is required only if the address description structure was created by *EvtChildListAddressDescriptionDuplicate* and contains driver-allocated dynamic memory that must be freed when the structure is deleted. The framework frees the structure itself.

Child-List Configuration Structure

The bus driver must initialize a child-list configuration structure with the size of the required child device identification description structure and then fill in information about the optional child device address description structure and related callback functions.

To configure the default child list

1. Call WDF_CHILD_LIST_CONFIG_INIT with the following parameters:
   - A pointer to a WDF_CHILD_LIST_CONFIG structure.
   - The size of the identification description structure.
   - A pointer to the *EvtChildListCreateDevice* callback function.
2. Set optional, additional information in the WDF_CHILD_LIST_CONFIG structure:
   - Pointers to the identification description callback functions.
   - The size of the address description structure.
   - Pointers to the address description callback functions.
   - A pointer to the *EvtChildListScanForChildren* callback function.
   - A pointer to the *EvtChildListDeviceReenumerated* callback function.
3. Call **WdfFdoInitSetDefaultChildListConfig**, which configures the default child list.

In the following example, a bus driver configures a child list that contains an identification description structure but does not contain an address description structure. The driver specifies an *EvtChildListCreateDevice* callback function that is named BusCreateChildDevice and an *EvtChildListScanForChildren* callback function that is named BusEnumerateBus:

```
WDF_CHILD_LIST_CONFIG_INIT (&childListConfig,
                            sizeof(ChildIdentificationDescription),
                            BusCreateChildDevice);
childListConfig.EvtChildListScanForChildren = BusEnumerateBus;
WdfFdoInitSetDefaultChildListConfig (DeviceInit,
                                     &childListConfig,
                                     WDF_NO_OBJECT_ATTRIBUTES);
```

The bus driver can then call **WdfDeviceCreate** to create the FDO for the parent bus. When the bus driver creates the FDO, the framework creates the default child list for the children of the parent bus. The lifetime of the FDO controls the lifetime of the child list because the WDFDEVICEOBJECT for the FDO is the parent object to the WDFCHILDLIST object. The bus driver must call the **WdfFdoGetDefaultChildList** method to get a handle to the default child list before it adds any children to the list.

If the bus driver requires any additional child lists, it can create them at any time after it has successfully created the FDO for the parent bus. Each additional child list must be configured before its creation, and each child list can be configured differently and with child list–specific callback functions.

To create an additional child list

1. Create the FDO for the parent bus.
2. Call WDF_CHILD_LIST_CONFIG_INIT with the following parameters:
   - A pointer to a WDF_CHILD_LIST_CONFIG structure.
   - The size of the identification description structure.
   - A pointer to the *EvtChildListCreateDevice* callback function.
2. Set the following optional, additional information in the WDF_CHILD_LIST_CONFIG structure:
   - Pointers to the identification description callback functions.
   - The size of the address description structure.
   - Pointers to the address description callback functions.
   - A pointer to the *EvtChildListScanForChildren* callback function.
   - A pointer to the *EvtChildListDeviceReenumerated* callback function.
3. Call **WdfChildListCreate** to create the additional child list. This method takes the following four parameters:
   - The handle to the FDO for the parent bus.
   - A pointer to the initialized WDF_CHILD_LIST_CONFIG structure.
   - A pointer to a WDF_OBJECT_ATTRIBUTES structure for the child-list object.
   - A pointer to a location in which the framework returns a handle to the newly created child-list object.

   The pointer to the attributes structure is optional. Attributes are typically not required for child-list objects. However, if you supply this parameter, the value of the **Parent** member of the structure must be NULL. The FDO of the parent bus is always the parent object of a child-list object, and a driver cannot change this attribute.

The following example shows how to create an additional child list:

```
WDF_CHILD_LIST_CONFIG_INIT( &ChildListConfig,
                            sizeof(CHILD_DEVICE_ID_DESCRIPTION),
                            &EvtCreateChild
                            );

WdfChildListConfig.AddressDescriptionSize =
                            sizeof(CHILD_DEVICE_ADDRESS_DESCRIPTION);

Status = WdfChildListCreate( BusDevice,
                             &ChildListConfig,
                             &ChildListAttributes,
                             OtherChildDevices
                             );
```

In the example, BusDevice is a handle to the parent bus FDO and ChildListConfig is a WDF_CHILD_LIST_CONFIG structure. The bus driver calls WDF_CHILD_LIST_CONFIG _INIT to initialize the child-list configuration structure with the size of the identification structure and a pointer to the *EvtChildListCreateDevice* callback function, which is named EvtCreateChild in the example. The driver then fills in the **AddressDescriptionSize** member. The call to **WdfChildListCreate** creates a child list

that has the specified configuration and returns the handle in the OtherChildDevices variable, which was declared as a pointer to a WDFCHILDLIST.

Reporting Child Devices to the Framework

After the bus driver configures the child list and creates the parent FDO for the parent bus, it can report children to the framework. As mentioned earlier, a bus driver should report children at the point in its processing at which it has access to the information that uniquely identifies each child device. Typically, this occurs after the parent bus is in the D0 state and its interrupts are connected.

If your bus driver implements the optional *EvtChildListScanForChildren* callback function, KMDF calls the driver every time that the parent bus enters D0 so that the driver can enumerate child devices. A bus driver is not required to implement this function; instead, it can enumerate devices from another place in its code. However, a bus driver must be able to enumerate children at every time that the bus reenters D0.

A bus driver that implements *EvtChildListScanForChildren* must register this callback in the WDF_CHILD_LIST_CONFIG structure before it calls **WdfFdoInitSetDefaultChildListConfig**, as follows:

```
WDF_CHILD_LIST_CONFIG_INIT(&config,
                           sizeof(PDO_IDENTIFICATION_DESCRIPTION),
                           MyBus_EvtDeviceListCreatePdo);

config.EvtChildListScanForChildren = MyBus_ScanForChildren;

WdfFdoInitSetDefaultChildListConfig(DeviceInit,
                                    &config,
                                    WDF_NO_OBJECT_ATTRIBUTES);
```

Regardless of whether a bus driver reports child devices from *EvtChildListScanForChildren*, *EvtDeviceD0Entry*, or some other function, the driver must use one or more of the **WdfChildListXxx** methods to populate and update the child list. Table 9 lists the methods for adding and removing devices from the child list.

Table 9. KMDF Methods that Update the Child List

Method	Description
WdfChildListAddOrUpdateChildDescriptionAsPresent	Adds a single child device to the child list or marks an existing child device as present.
WdfChildListBeginScan	Indicates the beginning of an update session and marks all devices in the child list missing.
WdfChildListEndScan	Indicates the end of an update session and reports the current contents of the child list to the PnP manager.
WdfChildListUpdateAllChildDescriptionsAsPresent	Marks all child devices in the child list as present.
WdfChildListUpdateChildDescriptionAsMissing	Marks a single child device in the child list as missing.

By using these methods in different combinations, you can efficiently add and remove child devices from the child list. To handle the addition or removal of several

children at the same time, a bus driver can use the **WdfChildListBeginScan** and **WdfChildListEndScan** methods to bracket the changes, in effect creating an update session. When the bus driver calls **WdfChildListBeginScan**, the framework marks all the devices in the child list as missing but does not report the missing child devices to the PnP manager. You can then call additional methods to add and remove devices from the child list as required. After you have made all the changes to the child list, a call to **WdfChildListEndScan** causes the framework to report the changes in the child list to the PnP manager.

To report one child device to the framework

1. Initialize an identification description structure for the child device by calling WDF_CHILD_IDENTIFICATION_DESCRIPTION_HEADER_INIT with a pointer to the WDF_CHILD_IDENTIFICATION_DESCRIPTION_HEADER structure and the size of the driver-defined identification description structure. Fill in the driver-defined members of the identification description structure.
2. If the child device requires an address description, initialize the address description structure by calling WDF_CHILD_ADDRESS_DESCRIPTION_HEADER _INIT with a pointer to the WDF_CHILD_ADDRESS_DESCRIPTION_HEADER structure and the size of the driver-defined child device address description structure. Fill in the driver-defined members of the address description.
3. Call **WdfChildListAddOrUpdateChildDescriptionAsPresent** and supply pointers to the identification description structure and the optional address description structure for the child device. This method notifies the framework that the bus driver has found a new child device. In response, the framework updates the child list and later calls the bus driver's *EvtChildLstCreateDevice* callback function to create a PDO for the child device.

To report several child devices or update the entire child list

During startup—and any other time that the bus driver enumerates several child devices—the bus driver can update the entire child list at the same time. To do this:

1. Call **WdfChildListBeginScan**.

   This method marks all the child devices in the child list as missing. Internally, the framework takes out a lock on the child list to protect against concurrent changes. No locks are required in the driver.
2. Initialize and fill in an identification description structure and, if appropriate, an optional address description structure, for the first child on the bus. Then report the child to the framework by calling **WdfChildListAddOrUpdateChildDescriptionAsPresent** and supplying pointers to the identification description structure and the optional address description structure.
3. Repeat step 2 for every additional child on the bus.
4. Call **WdfChildListEndScan**. In response, the framework reports the changes in the child list to the PnP manager and later calls the bus driver's *EvtChildListCreateDevice* callback function for each child device in the child list that does not already have a PDO.

These steps cause the framework to mark all the devices in the existing child list as missing. Therefore, the code that appears between the calls to **WdfChildListBeginScan** and **WdfChildListEndScan** must repopulate the entire list. The list is internal to the framework. Because the framework only notifies the PnP manager about new or changed devices, it does not send Plug and Play notifications or display dialog boxes about every device in the list.

To add or update a few child devices and otherwise keep all the existing child devices in the child list, call **WdfChildListUpdateAllChildDescriptionsAsPresent** after you call **WdfChildListBeginScan**. This method marks all the child devices present. You can then add or delete child devices as required.

The OsrFx2 EnumSwitches driver treats each switch on the OSR Fx2 test board as a separate child device and reports these devices to the framework as in the following example:

```
VOID
OsrFxEnumerateChildren(
    IN WDFDEVICE Device
    )
{
    WDFCHILDLIST    list;
    UCHAR           i;
    NTSTATUS        status;
    PDEVICE_CONTEXT pDeviceContext;

    pDeviceContext = GetDeviceContext(Device);

    list = WdfFdoGetDefaultChildList(Device);

    WdfChildListBeginScan(list);
    for(i=0; i< RTL_BITS_OF(UCHAR); i++) {
        //
        // Report every set bit in the switchstate as a child device.
        //
        if(pDeviceContext->CurrentSwitchState & (1<<i)) {
            PDO_IDENTIFICATION_DESCRIPTION description;
            WDF_CHILD_IDENTIFICATION_DESCRIPTION_HEADER_INIT(
                &description.Header,
                sizeof(description)
                );
            // Code omitted
            status = WdfChildListAddOrUpdateChildDescriptionAsPresent(
                            list,
                            &description.Header,
                            NULL); // AddressDescription
            if (status == STATUS_OBJECT_NAME_EXISTS) {
            // Error handling code goes here
            }
        }
    }
    WdfChildListEndScan(list);
    return;
}
```

In the example, the OsrFx2 EnumSwitches driver uses the default child list, so it calls **WdfFdoGetDefaultChildList** to get a handle to the default child list. Then it calls **WdfChildListBeginScan**. The call to **WdfChildListBeginScan** causes the framework to mark any children that are already in the child list as missing. Therefore, the driver must provide identification descriptions for all current children. For each switch, the

driver creates an identification description structure and calls WDF_CHILD_IDENTIFICATION_DESCRIPTION_HEADER_INIT to initialize the structure with the WDF-required header and size. The driver then calls **WdfChildListAddOrUpdateChildDescriptionAsPresent** to add the child device to the child list. When the loop ends, the driver calls **WdfChildListEndScan** to indicate that it has added all the current child devices to the child list.

Creating a PDO for a Dynamically Enumerated Child

After the bus driver reports a child device to the framework, the framework notifies the PnP manager, which initiates Plug and Play processing for the child device. As part of this processing, the framework calls the bus driver's *EvtChildListCreateDevice* callback function to create the PDO for the child device.

Creating a PDO in the dynamic enumeration model is the same as in the static enumeration model, with two important exceptions:

- A bus driver that performs dynamic enumeration must implement an *EvtChildListCreateDevice* callback function. This function must call **WdfDeviceCreate** to create a PDO for a child device.
- A bus driver is not required to create a WDFDEVICE_INIT structure. The framework allocates this structure and passes a pointer to it in the call to *EvtChildListCreateDevice*.

The parameters to *EvtChildListCreateDevice* are a handle to the child-list object, a pointer to the identification description structure that the driver provided when it reported the child device, and a pointer to a WDFDEVICE_INIT structure. The callback function must initialize the WDFDEVICE_INIT structure and call the framework to create the PDO for the child device, as described in "Creating a PDO for a Child Device" earlier in this paper. The framework does not call *EvtChildListCreateDevice* concurrently with other callbacks for the same device.

If the bus driver successfully creates the PDO, *EvtChildListCreateDevice* must return STATUS_SUCCESS or any other status for which NT_SUCCESS(*status*) equals TRUE. If PDO creation fails, the callback function must return a failure status. However, if an error occurs before the callback function calls **WdfDeviceCreate**, the bus driver can return STATUS_RETRY, which causes the framework to call *EvtChildListCreateDevice* again later. After three retries, the framework stops calling the callback function for that child device.

Device Removal in the Dynamic Enumeration Model

When the user removes a dynamically enumerated child device, the bus driver must update the child list to indicate that the child device is no longer present.

Procedures for these common tasks follow:

- Removing a single child device from the child list.
- Removing all child devices from the child list.
- Removing several child devices from the child list.

To remove a single child device from the child list

- Call **WdfChildListUpdateChildDescriptionAsMissing** and pass the handle to the child list and a pointer to the identification description structure for the child device.

  In response, the framework marks the child device as missing in the child list. After the method returns, the framework invalidates the bus relations for the child device's FDO and notifies any drivers that have registered for Plug and Play event notification that the child device is no longer present.

The following example shows how the Dynamic Toaster bus driver removes a single child device, given the serial number of the device to remove:

```
list = WdfFdoGetDefaultChildList(Device);

PDO_IDENTIFICATION_DESCRIPTION description;
WDF_CHILD_IDENTIFICATION_DESCRIPTION_HEADER_INIT(
        &description.Header,
        sizeof(description)
        );

description.SerialNo = SerialNo;
status = WdfChildListUpdateChildDescriptionAsMissing(list,
                                    &description.Header);
if (status == STATUS_NO_SUCH_DEVICE) {
    // Error handling code omitted;
}
```

In the example, the bus driver gets a handle to the default child list by calling **WdfFdoGetDefaultChildList**. The bus driver then initializes an identification description header so that it can fill in an identification description structure to pass to the child-list update method. It sets the serial number of the child device to be removed in the identification description structure and calls **WdfChildListUpdateChildDescriptionAsMissing** to request that the framework mark the child device that matches the identification description structure as missing. If the framework cannot find a matching child device, the method returns STATUS_NO_SUCH_DEVICE and the bus driver handles the error.

To remove all child devices from the child list

- Call **WdfChildListBeginScan**, followed by **WdfChildListEndScan**.

  WdfChildListBeginScan causes the framework to mark as missing all the children that are currently in the child list. When the bus driver calls **WdfChildListEndScan**, the framework reports the status of the child list to the PnP manager. After that method returns, the framework invalidates the bus relations for all the child device FDOs and notifies the drivers that have registered for Plug and Play event notification that the child devices are no longer present.

The following example shows how the Dynamic Toaster bus driver removes all child devices from the child list:

```
list = WdfFdoGetDefaultChildList(Device);
WdfChildListBeginScan(list);
WdfChildListEndScan(list);
```

To remove several child devices from the child list

- Use the **WdfChildListBeginScan** and **WdfChildListEndScan** methods to bracket an update session. **WdfChildListBeginScan** marks all child devices in the child as missing. Therefore, within the session you must call the **WdfChildListUpdateChildDescriptionAsPresent**, **WdfChildListUpdateAllChildDescriptionsAsPresent**, and **WdfChildListUpdateChildDescriptionAsMissing** methods as appropriate to mark devices present or missing. At the end of the session, **WdfChildListEndScan** reports the current status of the child list to the PnP manager, which proceeds as described earlier.

Finding Children in the Dynamic Child List

A bus driver that must periodically update the data about its child devices, such as a driver for an IEEE 1394 bus, can use KMDF methods to iterate through the child list without reenumerating all the devices. A bus driver can also retrieve the PDO for a specific child device.

KMDF defines the following methods for child list iteration:

- **WdfChildListBeginIteration**
- **WdfChildListRetrieveNextDevice**
- **WdfChildListEndIteration**

The first three methods in the list work basically the same as the similarly named methods for iterating through the static child devices. The primary difference is that dynamic child list iteration requires a WDF_CHILD_LIST_ITERATOR structure and can have an optional WDF_CHILD_RETRIEVE_INFO structure.

The WDF_CHILD_LIST_ITERATOR structure specifies which of the child devices to retrieve. A bus driver initializes this structure by calling the WDF_CHILD_LIST_ITERATOR_INIT function with two parameters: a pointer to the iterator structure and a WDF_RETRIEVE_CHILD_FLAGS enumeration value. These enumeration values are the same as those for static enumeration, as described in Table 6 in “Iterating through Static Children” earlier in this paper.

The WDF_CHILD_RETRIEVE_INFO structure provides locations into which the retrieval methods return the status of the child device and its identification description and address description. The status information is a value of the WDF_CHILD_LIST_RETRIEVE_DEVICE_STATUS enumeration, which provides additional detail about the status of the child device object. The bus driver should supply the WDF_CHILD_RETRIEVE_INFO structure if it requires either the status or the identification description before it calls additional framework methods. For example, the identification description is required in calls to the **WdfChildListRetrieveAddressDescription** method to get the address description.

To initialize the WDF_CHILD_RETRIEVE_INFO structure, a bus driver calls WDF_CHILD_RETRIEVE_INFO_INIT with a pointer to the WDF_CHILD_RETRIEVE_INFO structure and a pointer to a WDF_CHILD_IDENTIFICATION_HEADER structure. The bus driver must also initialize the WDF_CHILD_IDENTIFICATION_HEADER structure with the size of the information description structure for the child before it calls the framework to retrieve a child device from the list.

The following sample code shows how to iterate through a child list, retrieve the identification description for each child device, and update data for each retrieved child device:

```
// 1. Declare and initialize the iterator structure.
WDF_CHILD_LIST_ITERATOR       iterator;
WDF_CHILD_LIST_ITERATOR_INIT (&iterator,
                              WdfRetrievePresentChildren);
// 2. Start the search.
WdfChildListBeginIteration(list, &iterator);
    for ( ; ; ) {
// 3. Initialize child-device related structures
        WDF_CHILD_RETRIEVE_INFO         childInfo;
        PDO_IDENTIFICATION_DESCRIPTION  description;
        WDF_CHILD_RETRIEVE_INFO_INIT(&childInfo, &description.Header);
        WDF_CHILD_IDENTIFICATION_DESCRIPTION_HEADER_INIT(
                                &description.Header,
                                sizeof(description)
                                );
//4. Get the identification description for this child
        status = WdfChildListRetrieveNextDevice(list,
                                                &iterator,
                                                &hChild,
                                                &childInfo);
// 5. Handle failure and end of list.
        if (!NT_SUCCESS(status) || status == STATUS_NO_MORE_ENTRIES) {
                break;
        }
// 6. Child is not valid.
        ASSERT(childInfo.Status == WdfChildListRetrieveDeviceSuccess);
// 7. Do our job.
        // Use that description to do our task.
        // Code omitted

// 8. End the search.
WdfChildListEndIteration(list, &iterator);
```

The sample driver works as follows:

1. Initializes the iterator structure and specifies **WdfRetrievePresentChildren** to indicate that the framework should retrieve child devices that are currently marked as present in the child list. A driver could instead retrieve a different combination of present, pending, and missing child devices by specifying one of the other enumeration values from Table 6, earlier in this paper.
2. Calls **WdfChildListBeginIteration** to start searching through the list, beginning with the first child in the list.
3. Initializes a WDF_CHILD_RETRIEVE_INFO structure with a pointer to the header for the identification description structure for the child, and initializes the WDF_CHILD_IDENTIFICATION_DESCRIPTION_HEADER structure with the size of the identification description itself.
4 Calls **WdfChildListRetrieveNextDevice** to get the identification description for the next child device. This method returns the description in the previously initialized identification description structure.
5. Exits from the loop if the **WdfChildListRetrieveNextDevice** method fails or reaches the end of the list.
6. Checks the status of the child device object, which is returned in the **Status** member of the WDF_CHILD_RETRIEVE_INFO structure. The status is a value of

the WDF_CHILD_LIST_RETRIEVE_DEVICE_STATUS enumeration. If the retrieved child device does not have a corresponding framework device object, the status is set to a value other than **WdfChildListRetrieveDeviceSuccess** and consequently the bus driver ASSERTs.

7. Performs a task for each retrieved child device, which was edited from the example.
8. Calls **WdfChildListEndIteration** to end the search.

In addition to the methods that are shown in this example, KMDF provides two additional methods to retrieve information from a child list:

- **WdfChildListRetrievePdo** returns a handle to the PDO for the child that matches a specified identification description.
- **WdfChildListRetrieveAddressDescription** returns the address description for the child that matches a specified identification description.
- **WdfChildListGetDevice** returns a handle to the device object for the parent bus.

For more information about these methods, see “Framework Child-List Object Methods” in the WDK.

Reenumeration of Child Devices

If a child device fails, its function driver can request that the bus driver reenumerate the child device. KMDF defines a callback function in which a bus driver that performs dynamic enumeration can approve or veto reenumeration and can perform any required tasks before reenumeration occurs. In effect, the reenumerate-self functionality enables the drivers for a child device to rebuild its device stack without requiring user intervention. Self-reenumeration is supported only for dynamically enumerated child devices, and not for statically enumerated children.

The mechanism works as follows:

1. The function driver for a child device can request reenumeration in either of the following ways:
   - A KMDF function driver can call the **WdfDeviceSetFailed** method with the *FailedAction* parameter equal to **WdfDeviceFailedAttemptRestart**. Internally, the framework sends a Plug and Play IRP_MN_QUERY_INTERFACE request for the GUID_REENUMERATE_SELF_INTERFACE_STANDARD interface to the bus driver.
   - A KMDF or WDM driver can send a Plug and Play IRP_MN_QUERY_INTERFACE request for the GUID_REENUMERATE_SELF_INTERFACE_STANDARD interface.
2. On behalf of the bus driver, KMDF responds to the query with information about the reenumeration interface. KMDF implements the reenumeration interface for all framework-based drivers.
3. The child device's function driver invokes the reenumeration interface.
4. KMDF calls the bus driver’s *EvtChildListDeviceReenumerated* callback function. If the bus driver has not registered such a callback function, KMDF approves reenumeration and processing continues with step 6.

5. The bus driver's *EvtChildListDeviceReenumerated* callback function returns TRUE to approve reenumeration of the child device or FALSE to veto reenumeration of the child device. If *EvtChildListDeviceReenumerated* returns FALSE, processing ends with this step and the child list remains unchanged.
6. The framework marks the child device as not present and notifies the PnP manager that the child list has changed. The PnP manager consequently requests reenumeration of the child device.
7. The framework calls the bus driver's *EvtChildListCreateDevice* callback function later to create a new PDO for the child device.

The parameters to *EvtChildListDeviceReenumerated* are a handle to the child list, a handle to the PDO for the child device, and—if the child device has an address description—a pointer to the WDF_CHILD_ADDRESS_DESCRIPTION_HEADER structure for the existing address description for the child device and a pointer to a new address description header for the child device. The callback function should return TRUE to approve reenumeration of the child device or FALSE to cancel or veto reenumeration.

Usually, the bus driver simply approves reenumeration. However, a bus driver might veto reenumeration if it is aware of a problem with the device and can determine that reenumeration cannot resolve the problem.

If the callback function returns TRUE and the child device has an address description, the callback function must also return the new address information in the new address description.

The following example shows a simple implementation of this callback for a device that does not support address descriptions:

```
BOOLEAN
MyBus_EvtChildListDeviceReenumerated(
    WDFCHILDLIST  ChildList,
    WDFDEVICE  OldDevice,
    PWDF_CHILD_ADDRESS_DESCRIPTION_HEADER  OldAddressDescription,
    PWDF_CHILD_ADDRESS_DESCRIPTION_HEADER  NewAddressDescription
    )
{
    return TRUE;
}
```

Using a Raw PDO

A driver creates and uses a raw PDO in much the same way as a "regular" PDO does. The following are the important differences:

- A function or filter driver for the child device can create a raw PDO for the child device. Such a driver does not also act as the function driver for the parent bus.
- The driver must call **WdfPdoInitRawDevice** to notify the framework that the driver intends to manage the PDO raw. The driver must include a GUID that identifies the device setup class for the child device.
- If the PDO can service only one client at a time, the driver must call **WdfDeviceInitSetExclusive** to ensure that only one handle can be open to the PDO at a time.

- If the raw PDO is used for sideband communication and is created by a function or filter driver that can be loaded into several device stacks, the driver must ensure that it creates the raw PDO for the correct underlying physical device.
- If the raw PDO is used for sideband communication, the driver must provide a way for a client application to open handle to the PDO. The best way to provide such access is to create a device interface. However, if the client expects to use an MS-DOS® name for the device, the driver should instead create a symbolic link name. For information about how to create symbolic link names, see "**WdfDeviceCreateSymbolicLink**" in the WDK.

To determine the underlying physical device

A function or filter driver that can be loaded into more than one device stack must ensure that it creates the raw PDO for the correct physical device. KMDF calls a bus driver's *EvtDriverDeviceAdd* callback function for every device stack in which the driver is loaded. Therefore, this function must determine which device is at the bottom of the current stack. This information is stored in the registry. To retrieve information from the registry to determine the underlying device:

1. Call **WdfFdoInitAllocAndQueryProperty** before you create the FDO for the child device, or call **WdfDeviceAllocAndQueryProperty** after you create the FDO for the child device. Each of these methods allocates a buffer in the form of a WDFMEMORY object and retrieves a specified device property.

   The first parameter to **WdfFdoInitAllocAndQueryProperty** is a handle to the WDFDEVICE_INIT structure that was passed to the *EvtDriverDeviceAdd* callback function. The first parameter to **WdfDeviceAllocAndQueryProperty** is a handle to the FDO for the child device.

   Both methods take the following additional parameters:

   - A handle to the WDFDEVICE_INIT structure that was passed to *EvtDriverDeviceAdd*.
   - A value of the DEVICE_REGISTRY_PROPERTY enumeration that identifies the property to retrieve. Choose one or more properties that uniquely identify the device for which to create the raw PDO.
   - An enumeration of the POOL_TYPE value that specifies the type of memory to allocate for the returned memory object. The driver can use **NonPagedPool** because the framework always calls *EvtDriverDeviceAdd* at IRQL PASSIVE_LEVEL.
   - A pointer to a WDF_OBJECT_ATTRIBUTES structure that specifies the attributes for the memory object that the framework allocates.
   - A pointer to a WDFMEMORY location in which the framework returns a handle to the allocated memory object. On return from the method, the memory object contains the requested property.

2. Call **WdfMemoryGetBuffer** to get a pointer to the buffer that contains the value of the requested property.
3. If this is the device for which the driver requires a raw PDO, it can create the raw PDO.

To create a raw PDO

1. Allocate and initialize a WDFDEVICE_INIT structure for a PDO by calling **WdfPdoInitAllocate**.
2. Configure the PDO as a "raw" PDO by calling **WdfPdoInitRawDevice**. You must supply a class GUID so that the PDO can be started without a function driver.
3. Initialize other device properties that are appropriate for the device type and target operating system. These include a device ID and instance ID, which are required on all Windows versions. Other properties might include a hardware ID, security descriptor definition language (SDDL) string, locale ID, device text, and so on.
4. Create the PDO by calling **WdfDeviceCreate**.
5. Create I/O queues and set Plug and Play and power capabilities for the raw PDO as for any other device object.
6. Create a device interface by calling **WdfDeviceCreateDeviceInterface** so that a client application can open a handle to the PDO and send I/O requests to it.
7. Report the PDO to the framework.

The KbFiltr sample driver creates a raw PDO so that its application can communicate directly with a child device, without sending I/O requests through the overall Plug and Play device stack for the child device.

The following sample code shows how the KbFiltr sample creates a raw PDO:

```
    NTSTATUS                    status;
    PWDFDEVICE_INIT             pDeviceInit = NULL;
    PRPDO_DEVICE_DATA           pdoData = NULL;
    WDFDEVICE                   hChild = NULL;
    WDF_OBJECT_ATTRIBUTES       pdoAttributes;
    DECLARE_CONST_UNICODE_STRING(deviceId,KBFILTR_DEVICE_ID );
    DECLARE_CONST_UNICODE_STRING(hardwareId,KBFILTR_DEVICE_ID );
    DECLARE_UNICODE_STRING_SIZE(buffer, MAX_ID_LEN);
    //
    // 1. Allocate a WDFDEVICE_INIT structure and set the properties
    // so that we can create a device object for the child.
    //
    pDeviceInit = WdfPdoInitAllocate(Device);
    if (pDeviceInit == NULL) {
        status = STATUS_INSUFFICIENT_RESOURCES;
        goto Cleanup;
    }

    // 2.  Mark the device RAW so that the child device can be
    // started and accessed without a function driver. Because we are
    // creating a RAW PDO, we must provide a class guid.
    status = WdfPdoInitAssignRawDevice(pDeviceInit,
                                       &GUID_DEVCLASS_KEYBOARD);
    if (!NT_SUCCESS(status)) {
        goto Cleanup;
    }

    // 3.  Initialize device properties.
    status = WdfDeviceInitAssignSDDLString(pDeviceInit,
                    &SDDL_DEVOBJ_SYS_ALL_ADM_ALL);
    if (!NT_SUCCESS(status)) {
        goto Cleanup;
    }
```

```
    status = WdfPdoInitAssignDeviceID(pDeviceInit, &deviceId);
    if (!NT_SUCCESS(status)) {
        goto Cleanup;
    }
    status =  RtlUnicodeStringPrintf(&buffer, L"%02d", InstanceNo);
    if (!NT_SUCCESS(status)) {
        goto Cleanup;
    }
    status = WdfPdoInitAssignInstanceID(pDeviceInit, &buffer);
    if (!NT_SUCCESS(status)) {
        goto Cleanup;
    }

    //. . . Additional initialization code omitted

    // 4. Initialize the attributes to specify the size of the device
    // context area and the create the PDO.
    WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&pdoAttributes,
                                            RPDO_DEVICE_DATA);
    status = WdfDeviceCreate(&pDeviceInit, &pdoAttributes, &hChild);
    if (!NT_SUCCESS(status)) {
        goto Cleanup;
    }

    // 5. Create I/O queues.
    // Set Plug and Play and power capabilities.
    // Code omitted...

    // 6. Create a device interface so that
    // application can find the device and talk to it.
    //
    status = WdfDeviceCreateDeviceInterface(
                hChild,
                &GUID_DEVINTERFACE_KBFILTER,
                NULL
             );

    if (!NT_SUCCESS (status)) {
        DebugPrint( ("WdfDeviceCreateDeviceInterface failed 0x%x\n",
                      status));
        goto Cleanup;
    }

    //
    // 7. Report the device to the framework.
    status = WdfFdoAddStaticChild(Device, hChild);
    if (!NT_SUCCESS(status)) {
        goto Cleanup;
    }

    //
    // pDeviceInit will be freed by WDF.
    //
    return STATUS_SUCCESS;

Cleanup:

    // Call WdfDeviceInitFree to deallocate the initialization
    // structure if an error occurs. Do not call WdfDeviceCreate
    // after this.
    if (pDeviceInit != NULL) {
        WdfDeviceInitFree(pDeviceInit);
    }
```

```
    if(hChild) {
        WdfObjectDelete(hChild);
    }

    return status;
}
```

In addition to the standard steps for creating a PDO, note the following in the sample code:

- The driver reports the PDO to the framework by calling **WdfFdoAddStaticChild**. For more information on this method, see "Static Enumeration" earlier in this paper.
- The driver includes cleanup code that runs in case of errors. If an error occurs after allocation of the WDFDEVICE_INIT structure but before successful return from **WdfDeviceCreate**, the driver must call **WdfDeviceInitFree** to deallocate the structure. If an error occurs after **WdfDeviceCreate** has returned successfully, the framework has already deleted the device initialization structure, but the driver must delete the child PDO. All drivers should include code that cleans up after initialization errors.

Optional PDO Event Callback Functions

A bus driver can provide optional event callback functions for a child-device PDO to handle bus-level power-management requests, to handle resource requirements queries and, if applicable, to support ejection of the device. Table 10 lists these callback functions.

Table 10. Optional PDO Callback Functions

Function	Description
EvtDeviceEnableWakeAtBus	Enables a wake signal at bus level.
EvtDeviceDisableWakeAtBus	Disables a wake signal at bus level.
EvtDeviceResourceRequirementsQuery	Returns a list of the hardware resources that the child device requires.
EvtDeviceResourcesQuery	Returns a list of the boot configuration resources that the child device requires.
EvtDeviceSetLock	Locks or unlocks the child device to enable or prevent ejection.
EvtDeviceEject	Ejects a child device.

A bus driver that provides one or more of these callback functions must register them with the framework as part of device initialization, as described previously in "Device Initialization Structure."

To register the callback functions, a driver initializes a WDF_PDO_EVENT_CALLBACKS structure and then calls **WdfPdoInitSetEventCallbacks** to record the callback information in the WDFDEVICE_INIT structure before it creates the PDO for the child device. The following example shows how a bus driver registers the *EvtDeviceResourceRequirementsQuery*, *EvtDeviceEnableWakeAtBus*, and *EvtDeviceDisableWakeAtBus* callback functions:

```
WDF_PDO_EVENT_CALLBACKS_INIT(&pdoCallbacks);
pdoCallbacks.EvtDeviceResourceRequirementsQuery =
     Bus_Pdo_EvtDeviceResourceRequirementsQuery;
pdoCallbacks.EvtDeviceEnableWakeAtBus =
     Bus_Pdo_EvtDeviceEnableWakeAtBus;
pdoCallbacks.EvtDeviceDisableWakeAtBus =
     Bus_Pdo_EvtDeviceDisableWakeAtBus;
WdfPdoInitSetEventCallbacks(DeviceInit, &pdoCallbacks);
```

Bus-Level Power-Management Support

The bus driver is typically responsible for enabling and disabling a bus-level wake signal for a child device and for notifying the framework when a child device triggers a wake signal. If your bus driver must enable such a signal, you must implement the following pair of callback functions:

- *EvtDeviceEnableWakeAtBus*
- *EvtDeviceDisableWakeAtBus*

When the function driver enables or disables wake for the device, the framework calls these functions so that the bus driver can perform bus-level tasks that are related to the wake signal.

Enabling and Disabling a Wake Signal at the Bus

The function driver for a child device performs most of the work that is involved in enabling the device to trigger a wake signal. The bus driver, however, might be required to enable the wake signal at the bus, depending on the design of the particular bus or device. For example, the PCI bus driver must enable the wake signal on the bus before a PCI device can wake the system.

The PnP manager, framework, and drivers cooperate as follows to enable a wake signal for a child device:

1. The function driver for the child device enables the device's ability to wake.
2. When the device is powered down, the framework calls the bus driver's *EvtDeviceEnableWakeAtBus* callback function at the beginning of the shutdown sequence, while the child device is still in the D0 state. In this callback function, the bus driver must do whatever is required at the bus level to enable the wake signal.
3. When the child device returns to D0, the actions depend on whether the device triggered a wake signal:
    - If the child device triggered a wake signal, the system and the framework return the device to D0. The framework calls the bus driver's *EvtDeviceDisableWakeAtBus* callback function during startup of the child device. In this callback function, the bus driver should do whatever is required at the bus level to disable the wake signal, so that the device can no

longer trigger it. Thus, *EvtDeviceDisableWakeAtBus* reverses the actions of *EvtDeviceEnableWakeAtBus*.

- If the child device did not trigger a wake signal while it was powered down, the framework calls *EvtDeviceEnableWakeAtBus* during startup so that the bus driver can reenable the wake signal. Otherwise, wake remains disabled unless the function driver enables it.

When the child device is removed, the framework also calls *EvtDeviceDisableWakeAtBus*.

Note: If you have previously developed WDM drivers, you might suspect that the reason for the different restart behaviors is the underlying WDM IRP_MN_WAIT_WAKE request. Every time that the device triggers a wake signal, its drivers complete such a request. If the device does not trigger the signal, the request remains pending, and therefore the bus driver must reenable the wake signal.

Notifying the Framework about a Wake Signal

When one of its child devices triggers a wake signal, the bus driver must notify the framework about the wake event. In response, the framework initiates the startup sequence for the other drivers in the device stack for the child device, so that the child device can return to the working state.

Typically, a child device generates an interrupt to signal a wake event. In the *EvtInterruptDpc* callback function that processes the interrupt, the bus driver should call **WdfDeviceIndicateWakeStatus** to notify the framework that the child device triggered the wake signal. In response, the framework returns the child device to the working state.

Reporting Resources

The PnP manager sends two different types of resource queries to which bus drivers can respond:

- Resource requirements queries
- Resources queries

Resource Requirements Queries

The PnP manager sends a resource requirements query to obtain a list of hardware resource requirements for a device. The bus driver provides this list for its child devices, and the PnP manager uses the list as a guide when it assigns hardware resources to the child device. A bus driver that must respond to such queries implements an *EvtDeviceResourceRequirementsQuery* callback function. Only bus drivers that manage actual physical hardware should implement this callback function. Root-enumerated software devices, such as the sample Toaster bus, cannot request hardware resources for their child devices.

When the framework receives an IRP_MN_RESOURCE_REQUIREMENTS_QUERY from the PnP manager, the framework calls the bus driver's *EvtDeviceResourceRequirementsQuery* callback function with a handle to the PDO for the child device and a handle to an empty I/O resource requirements list. The callback function responds to the query by:

- Creating a logical resource configuration, which contains a range of resources that the child device can use.
- Adding the logical resource configuration to the empty I/O resource requirements list.
- Populating the logical resource configuration with one or more resource descriptors, each of which describes a particular hardware resource.

Logical resource configurations and resource descriptors are basically the same in KMDF drivers and WDM drivers. For more information, see "Hardware Resources for Framework-Based Drivers" and "IO_RESOURCE_DESCRIPTOR" in the WDK.

The following example shows how a bus driver can implement an *EvtDeviceResourceRequirementsQuery* callback function:

```
NTSTATUS
Bus_Pdo_EvtDeviceResourceRequirementsQuery(
    IN WDFDEVICE Device,
    IN WDFIORESREQLIST RequirementsList
    )
{
    IO_RESOURCE_DESCRIPTOR descriptor;
    NTSTATUS status;
    WDFIORESLIST reslist;
    UNREFERENCED_PARAMETER(Device);

// 1. Create a configuration for our list of resources.
    status = WdfIoResourceListCreate(RequirementsList,
                                     WDF_NO_OBJECT_ATTRIBUTES,
                                     &reslist);
    if (!NT_SUCCESS(status)) {
        return status;
    }
 // 2. Add the configuration to the empty list we got.
    status = WdfIoResourceRequirementsListAppendIoResList(
                RequirementsList, reslist);
    if (!NT_SUCCESS(status)) {
        return status;
    }
// 3. Set up resource descriptors.
    RtlZeroMemory(&descriptor, sizeof(descriptor));
    descriptor.Option = 0;
    descriptor.Type = CmResourceTypePort;
    descriptor.ShareDisposition = CmResourceShareDeviceExclusive;
    descriptor.Flags =
              CM_RESOURCE_PORT_IO|CM_RESOURCE_PORT_16_BIT_DECODE;
    descriptor.u.Port.Length = 1;
    descriptor.u.Port.Alignment = 0x01;
    descriptor.u.Port.MinimumAddress.QuadPart = 0;
    descriptor.u.Port.MaximumAddress.QuadPart = 0xFFFF;
    status = WdfIoResourceListAppendDescriptor(reslist, &descriptor);
    ASSERT(NT_SUCCESS(status));
```

```
    RtlZeroMemory(&descriptor, sizeof(descriptor));
    descriptor.Option = 0;
    descriptor.Type   = CmResourceTypeInterrupt;
    descriptor.ShareDisposition = CmResourceShareDeviceExclusive;
    descriptor.Flags  = CM_RESOURCE_INTERRUPT_LEVEL_SENSITIVE;
    descriptor.u.Interrupt.MinimumVector = 0;
    descriptor.u.Interrupt.MaximumVector = 15;
    descriptor.u.Interrupt.AffinityPolicy = IrqPolicyMachineDefault;
    descriptor.u.Interrupt.PriorityPolicy = IrqPriorityNormal;
    status = WdfIoResourceListAppendDescriptor(reslist, &descriptor);
    ASSERT(NT_SUCCESS(status));
// 4. Success!
    return status;
}
```

In the example, the bus driver requests two types of resources—a port resource and an interrupt resource—and works as follows:

1. Calls **WdfIoResourceListCreate** to create a logical resource configuration. If the method fails, the driver returns the failure status to the framework.
2. Calls **WdfIoResourceRequirementsListAppendIoResList** to add the logical resource configuration to the empty resource requirements list that the framework passed as a parameter. If the method fails, the driver returns the failure status to the framework.
3. Sets up each resource descriptor and calls **WdfIoResourceListAppendDescriptor** to add the resource descriptor to the resource configuration. If the method fails, the driver asserts.

   Each resource descriptor is an IO_RESOURCE_DESCRIPTOR structure that contains information about a particular type of hardware resource. After filling in the members that apply to the resource, the bus driver appends the descriptor to the resource configuration, which is already part of the resource requirements list.
4. Returns a status that indicates to the framework whether it successfully responded to the query.

Resources Queries

The PnP manager sends a resources query to obtain a list of the child device's boot configuration resources. These are the hardware resources that the system firmware assigned to the child device at system startup. The bus driver can get this information from the firmware. A bus driver that must respond to such queries must implement an *EvtDeviceResourcesQuery* callback function.

Supporting Ejectable Devices

A bus driver that supports a bus that has ejectable child devices, such as a laptop in a docking station, must set the **EjectSupported** member in the child device's WDF_DEVICE_PNP_CAPABILITIES structure during initialization, after it creates the PDO. In addition, the bus driver can provide one or both of the following optional callback functions to handle ejection:

- *EvtDeviceEject*
- *EvtDeviceSetLock*

Device Ejection

When the user presses a button to eject a child device, the device typically generates an interrupt. In response, the bus driver's *EvtInterruptDpc* callback function notifies the framework of the device eject event by calling either **WdfPdoRequestEject** or **WdfChildListRequestChildEject**.

The **WdfPdoRequestEject** method requires a handle to the PDO for the child device that is being ejected; drivers that perform static enumeration must use this method. If the driver has a pointer to the identification description for the child device instead of a handle to the PDO, it calls **WdfChildListRequestChildEject**.

When a bus driver calls either eject method, the framework informs the PnP manager that the child device is being ejected. The PnP manager performs the tasks that are required for an orderly ejection. Such tasks include notifying the other drivers in the child's device stack that the device is being removed and notifying the drivers for the child device's ejection relations—any other devices that are ejected together with the child device—that their devices are being removed. The framework performs an orderly removal of the child device by transitioning the device out of the D0 state, calling the bus driver's *EvtDeviceReleaseHardware* callback function and then calling the bus driver's *EvtDeviceEject* callback function so that the bus driver can perform any additional bus-level tasks that are required to eject the child device.

Both the Static Toaster and Dynamic Toaster bus drivers support child device ejection by using an IOCTL request instead of an interrupt to initiate the ejection of a child device. When the bus driver receives the ejection IOCTL, the driver iterates through the child list and removes the specified child device. The following example shows how the Static Toaster bus driver ejects a child device:

```
NTSTATUS
Bus_EjectDevice(
    WDFDEVICE   Device,
    ULONG       SerialNo
    )

{
    PPDO_DEVICE_DATA pdoData;
    BOOLEAN          ejectAll;
    WDFDEVICE        hChild;
    NTSTATUS         status = STATUS_INVALID_PARAMETER;

    PAGED_CODE ();

/*  SerialNo indicates device to eject. 0 means eject all.
    ejectAll = (0 == SerialNo) ? TRUE : FALSE;
    hChild = NULL;

    WdfFdoLockStaticChildListForIteration(Device);

    while ((hChild = WdfFdoRetrieveNextStaticChild(Device,
                         hChild,
                          WdfRetrieveAddedChildren)) != NULL) {
        pdoData = PdoGetData(hChild);
/*      Eject this device?
        if (ejectAll || SerialNo == pdoData->SerialNo) {
            status = STATUS_SUCCESS;
            WdfPdoRequestEject(hChild);
            if (!ejectAll) {
                break;
```

```
            }
        }
    }
    WdfFdoUnlockStaticChildListFromIteration(Device);
    return status;
}
```

Device ejection in the Dynamic Toaster bus driver is similar, except that it uses the **WdfChildListBeginIteration**, **WdfChildListRetrieveNextDevice**, and **WdfChildListEndIteration** methods to iterate through the child list.

The framework deletes the PDO. The bus driver must not delete the PDO.

Device Locking

If an ejectable child device can be locked in place, the bus driver must set the **LockSupported** member in the child device's WDF_DEVICE_PNP_CAPABILITIES structure during initialization, after it creates the PDO. The bus driver must also supply an *EvtDeviceSetLock* callback function. When the framework receives an IRP_MN_SET_LOCK request for the child device from the PnP manager, it calls *EvtDeviceSetLock* so that the bus driver can lock the device to prevent ejection or can unlock the device to enable ejection.

EvtDeviceSetLock should perform whatever tasks the child device hardware requires to lock or unlock the device.

Handling I/O Requests for the PDO

The bus driver is responsible for handling I/O requests that are directed to the PDOs for the child devices that it enumerates. In most cases, the bus driver sets up I/O queues, receives I/O requests, and completes I/O requests exactly like a function or filter driver. However, the following situations require bus driver–specific code:

- Forwarding requests to the FDO for the parent bus.
- Self-managed I/O flush and cleanup when the child device is removed.
- Synchronization of callback functions during power-up and removal of a child device.

Forwarding an I/O Request to the FDO for the Parent Bus

The PDO is the final destination for most I/O requests. Therefore, when an I/O request arrives for the PDO of a child device, the bus driver typically performs the operation and completes the request. However, a bus driver must occasionally forward certain I/O requests to the FDO for the parent bus. For example, a protocol driver for a USB hub must forward some requests to the next USB hub or controller stack for processing.

Drivers that are released with KMDF version 1.9 can use framework methods to forward requests in this situation. Drivers that are released with earlier versions must manually forward such requests.

Forwarding Requests in KMDF Version 1.9 and Later

KMDF version 1.9 includes methods to forward requests to the FDO for the parent bus. A bus driver that expects to forward such requests must inform the framework by calling **WdfPdoInitAllowForwardingRequestToParent** when it creates the child

PDO. It can later call **WdfRequestForwardToParentDeviceIoQueue** to forward a request.

Forwarding Requests in Earlier KMDF Versions

Versions of KMDF earlier than 1.9 do not support methods that forward requests on behalf of the driver. Instead, the driver must include code to manually forward requests.

To forward a request to the parent bus FDO, a bus driver must create and open a WDF I/O target that represents the parent bus FDO, share the I/O target with the child device object, and adjust the stack size in the child device object. When an I/O request arrives, the driver formats the request for the I/O target and then sends the request. This mechanism provides a single I/O target object for all child devices, correctly sets up the underlying request, and ensures that the lifetime of the I/O target object matches that of the parent bus FDO.

The bus driver must use an I/O target to ensure that the underlying I/O request packet (IRP) is correctly set up. The driver must not simply call the corresponding routines for the FDO, even though the bus driver is also the function driver for the parent bus—and therefore contains the code that handles I/O requests for the parent bus FDO. If the bus driver bypasses the usual forwarding mechanism and the request is later forwarded from the parent FDO to a lower driver in the parent stack, the underlying IRP does not contain enough stack locations for the completed request to unwind correctly through all the drivers that have handled it and the system crashes.

The code to create and open the I/O target typically appears in the bus driver's *EvtDriverDeviceAdd* callback function for the parent FDO. The bus driver cannot share the I/O target handle and adjust the stack size until after it has created the child PDO. Therefore, the code to perform those tasks depends on whether the driver supports static or dynamic enumeration.

To create and open an I/O target that represents the parent bus FDO

1. After you create the FDO for the parent bus, get a pointer to the WDM device object for the parent bus FDO by calling **WdfDeviceWdmGetDeviceObject** and passing a handle to the WDF device object for the parent bus FDO.
2. Create an I/O target object by calling **WdfIoTargetCreate** and passing a handle to the parent bus FDO. This method creates a WDFIOTARGET object that is a child object of the parent bus FDO.
3. Initialize a WDF_IO_TARGET_OPEN_PARAMS structure for the parent FDO by calling WDF_IO_TARGET_OPEN_PARAMS_INIT_EXISTING_DEVICE and passing the pointer to the WDM device object for the parent bus FDO. This function initializes the members of the structure that indicate that the I/O target is an existing WDM device object.
4. Open the I/O target object by calling **WdfIoTargetOpen** and passing the handle to the I/O target object and a pointer to the initialized WDF_IO_TARGET_OPEN_PARAMS structure.
5. Save the handle to the I/O target object in the device context area of the parent bus FDO.

The following example shows how to perform these steps:

```
// Step 1. Get a pointer to the WDM device object for parent bus
fdoDeviceObject = WdfDeviceWdmGetDeviceObject(
                  ParentDeviceContext->Device); // Parent bus FDO

//Step 2. Create the I/O target object.
status = WdfIoTargetCreate (ParentDeviceContext->Device,
                            WDF_NO_OBJECT_ATTRIBUTES,
                            &ioTarget);
if (!NT_SUCCESS (status)) {
    return status;
}

//Step 3. Initialize the open params structure.
WDF_IO_TARGET_OPEN_PARAMS_INIT_EXISTING_DEVICE (&openParams,
                                                fdoDeviceObject);

//Step 4. Open the I/O target.
status = WdfIoTargetOpen (ioTarget, &openParams);
if (!NT_SUCCESS (status)) {
    return status;
}

//Step 5. Save the I/O target handle.
ParentDeviceContext ->IoTargetToSelf = ioTarget;
```

In the example, ParentDeviceContext is a pointer to the device context area for the parent bus FDO and ParentDeviceContext->Device contains a handle to the WDF device object for the parent bus FDO. The steps in the sample typically would appear in the bus driver's *EvtDriverDeviceAdd* for the parent bus.

After the driver enumerates a child device and creates a PDO for it, the driver can share the I/O target with the child and adjust the stack size.

To share the I/O target with a child PDO and adjust the stack size

1. After you create the PDO for a child device, save the I/O target handle in the context area of the child-device PDO.
2. Adjust the size of the I/O stack in the child device object to account for the size of the I/O stack of the parent bus.

   This step is required because the bus driver forwards the I/O request from the child-device PDO to a different device stack—that of the parent bus FDO. The I/O request must have one I/O stack location for each driver that handles it.

   In KMDF versions 1.7 and earlier, the framework does not provide an interface to modify the I/O stack size. Instead, the bus driver must access the WDM device objects and do this itself. To do this, the bus driver must call **WdfDeviceWdmGetDeviceObject** to get a pointer to the WDM device object for the child device (the PDO) and then set the I/O stack size in the PDO equal to the I/O stack size in the parent bus FDO plus one. The I/O stack size is stored in the **StackSize** member of the WDM device object.

The following example shows how to save the I/O target handle and adjust the I/O stack size:

```
ChildDeviceContext->IoTargetToParentBusFdo
          = ParentDeviceContext->IoTargetToSelf;
pdoDeviceObject = WdfDeviceWdmGetDeviceObject(ChildDevice);
pdoDeviceObject->StackSize = fdoDeviceObject->StackSize + 1;
```

After these steps are complete, the child-device PDO has an I/O target that represents the parent bus FDO. When the child-device PDO receives an I/O request that must be forwarded to the parent bus FDO, the driver calls one of the **WdfIoTargetFormatXxx** or **WdfIoTargetSendXxx** methods as appropriate to format the I/O request and then send it to the I/O target.

Self-Managed I/O Flush and Cleanup

The self-managed I/O interface in KMDF enables drivers to handle requests that the framework does not manage, such as IRPs that communicate with their devices or with other drivers in the device stack at particular points in the device startup and shutdown sequences. Bus drivers often use self-managed I/O to implement watchdog timers.

If your bus driver implements self-managed I/O callback functions for a PDO, you must be aware of exactly when the framework calls the bus driver's *EvtDeviceSelfManagedIoFlush* and *EvtDeviceSelfManagedIoCleanup* callback functions and what tasks these callback functions should perform. The framework calls these functions only during child device removal. It does not call them if the child device is transitioning to a low-power state or if the PnP manager is stopping the system to rebalance resources. Figure 3 summarizes the order of self-managed I/O operations in the child-device removal sequence.

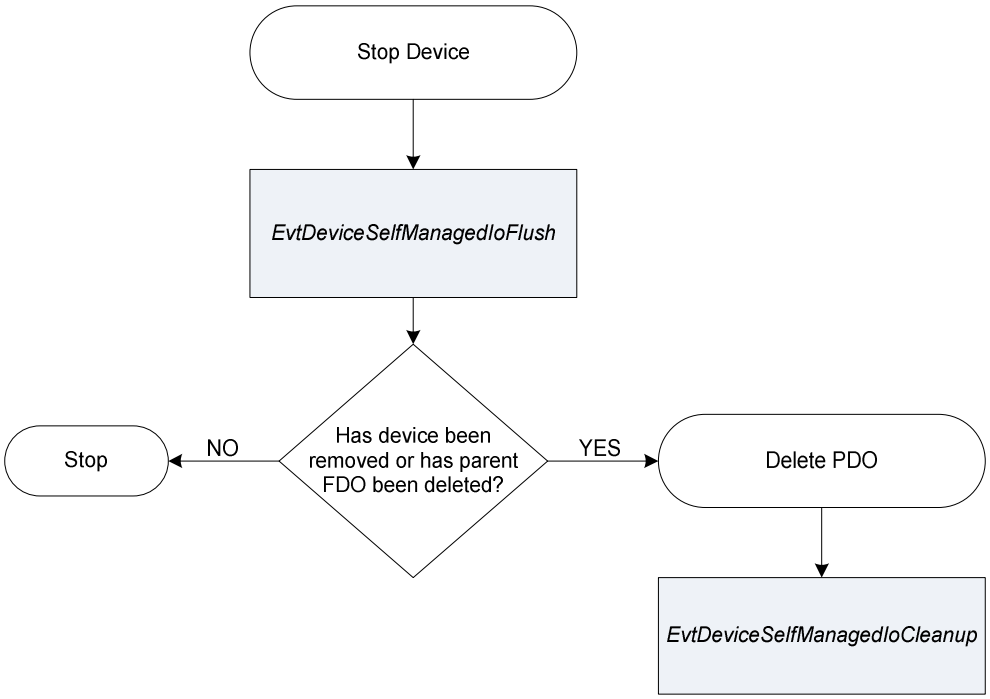

Figure 3. Self-managed I/O calls during PDO device removal

In Figure 3, shaded boxes represent driver code and unshaded boxes represent framework processing.

After the framework stops the child device, it calls the bus driver's *EvtDeviceSelfManagedIoFlush* callback function. In this function, the bus driver should fail any remaining incomplete self-managed I/O requests for the child device and save any information that it requires later to restart the child device. When the framework calls *EvtDeviceSelfManagedIoFlush*:

- The bus driver cannot access the hardware for the child device because the child device is in the D3 power state.
- The bus driver's *EvtDeviceReleaseHardware* callback function has already run. Therefore, the bus driver cannot access any resources that *EvtDeviceReleaseHardware* released.
- All the child device's power-managed queues have been stopped.
- Both the WDF and WDM device objects for the PDO still exist.

At this point, the framework's device-removal processing for the child device ends if the child device is still physically present in the system—for example, if the user has disabled it in Device Manager but has not removed the hardware. The framework retains the PDO. If the user later restarts or reenables the child device, the framework calls the bus driver's *EvtDeviceSelfManagedIoRestart* callback function. In this situation, the bus driver's *EvtDeviceSelfManagedIoCleanup* callback function is never called for the child device.

If the child device was physically removed or the parent bus was deleted, the framework's device removal processing continues after *EvtDeviceSelfManagedIoFlush* returns. The framework deletes the WDM PDO for the child device and calls the bus driver's *EvtDeviceSelfManagedIoCleanup* callback function. This function should release any resources that the driver allocated specifically for use during self-managed I/O for the child device. Such allocation typically occurs in *EvtDeviceSelfManagedIoInit* or *EvtDeviceSelfManagedIoRestart*. When the framework calls the bus driver's *EvtDeviceSelfManagedIoCleanup* callback function:

- The bus driver cannot access the hardware for the child device because the child device is no longer physically present.
- All child device queues that are not power managed have been stopped.
- The WDF device object for the PDO still exists but the WDM device object has been deleted.

After *EvtDeviceSelfManagedIoCleanup* returns, the framework deletes the WDF device object for the child device and calls the bus driver's *EvtCleanupCallback* and *EvtDestroyCallback* callback functions so that the bus driver can free any resources that it allocated for the device context area for the child-device PDO.

Synchronization of Power Callbacks

If the parent bus FDO is the PPO for its device stack, KMDF synchronizes power callbacks to parent bus and child device objects so that the parent bus always has power when its child devices have power.

When a child device is powered on (in the D0 state), the parent bus is also powered on. The parent bus must be in the D0 state before any of its child devices can enter the D0 state. The framework synchronizes calls to the bus driver's power-up callbacks so that all the power callbacks for the parent bus run before those of any of its child devices. The parent bus and its function driver must be operational before any of the child devices on the bus can operate.

Conversely, when the parent bus is powered down (in a D*x* power state) or removed, all the child devices on the bus must also be powered down or removed. The framework synchronizes device power-down and removal so that the power callbacks for each child device on the bus run before those of the parent bus. In this situation, the parent bus must remain operational until all its child devices have left the working state.

Therefore, if the parent bus FDO owns power policy, the bus driver's power-management callbacks for the parent bus FDO do not require additional synchronization to prevent conflicts with those of the child-device PDOs. However, the framework's synchronization applies only to the power policy owner. Therefore, if the driver that creates the PDO is not the power policy owner—as in the case of a raw PDO that is used for sideband communication— you can make no assumptions about synchronization of the power callbacks between the parent bus and its child devices.

Handling Device Errors

If a bus driver determines that a child device is no longer responding, it should update the child list or notify the framework, as appropriate. Specifically:

- If one of the child devices has become inaccessible, the bus driver should mark the device as missing in the child list by calling **WdfPdoMarkMissing** or **WdfChildListUpdateChildDescriptionAsMissing**.
- If the child device is still accessible but is unusable and does not respond to commands, the bus driver should set the state of the child device to Failed. To do this, the driver calls **WdfDeviceSetDeviceState** and passes a WDF_DEVICE_STATE structure that has the **Failed** member set to **WdfTrue**. The framework notifies the PnP manager, which marks the device with an error in Device Manager.

Tips for Testing and Debugging a Bus Driver

Several WDK tools and features are especially useful with bus drivers:

- Plug and Play Driver test.
- The **!wdfchildlist** debugger extension.
- The KMDF log.

The Plug and Play Driver test (Pnpdtest.exe) exercises Plug and Play drivers by sending Plug and Play requests down the device stack. This test provides special options for testing graceful removal, surprise removal, double start, and resource rebalancing, so that you can ensure that your driver handles all these operations correctly. To obtain the most thorough results, you should run Pnpdtest with Driver Verifier enabled. For more information about Pnpdtest, see "Plug and Play Driver Test" in the WDK.

The Microsoft kernel-mode debuggers support a package of KMDF-specific extension commands, which display information about WDF objects. The **!wdfchildlist** extension is especially useful in debugging a bus driver because it provides information about the contents of a child list, such as the identification descriptions and the callback functions that are registered for the child devices in the list. For complete information about the KMDF-specific extensions, see "Kernel-Mode Driver Framework Extensions" on MSDN.

As you test your driver, be sure to consult the KMDF log. The log tracks the progress of I/O requests through the driver and the framework and records when the framework reports each new child-device PDO to the PnP manager. For detailed information about how to view the log, see "How to Use the KMDF Log" on the WHDC Web site.

In all other respects, testing and debugging a KMDF bus driver are no different from testing and debugging any other KMDF driver, and the same guidelines apply. Specifically:

- Enable tracing in your driver and include trace statements to log errors and other significant data.
- Use Static Driver Verifier with KMDF-specific rules to check for code that violates KMDF driver rules.
- Run PRE*f*ast for Drivers to simulate execution of code paths.
- Enable both Driver Verifier and KMDF Verifier during testing.

For more information about these tools, see "Tools for Software Tracing" and "Tools for Verifying Drivers" in the WDK. In addition, *Developing Drivers with the Windows Driver Foundation* contains specific examples and guidelines for using these tools with KMDF drivers.

Resources

WHDC Web site

Windows Driver Foundation (WDF) on the WHDC Web site
http://www.microsoft.com/whdc/driver/wdf/default.mspx

Plug and Play—Architecture and Driver Support on the WHDC Web site
http://go.microsoft.com/fwlink/?LinkId=82116

Books

Developing Drivers with the Windows Driver Foundation, by Penny Orwick and Guy Smith
http://www.microsoft.com/MSPress/books/10512.aspx

White Papers and Driver Tips on the WHDC Web site

How to Use the KMDF Log
http://www.microsoft.com/whdc/driver/tips/KMDF_IfrLog.mspx

No time to write a bus driver? Try using a device object namespace.
http://www.microsoft.com/whdc/driver/tips/DevNamespace.mspx

WDK documentation

Enumerating the Devices on a Bus
http://msdn2.microsoft.com/en-us/library/aa490094.aspx

Framework Child-List Object Methods
http://msdn.microsoft.com/en-us/library/aa490670.aspx

General Framework Device Object Initialization Methods
http://msdn.microsoft.com/en-us/library/aa490990.aspx

Hardware Resources for Framework-Based Drivers
http://msdn.microsoft.com/en-us/library/aa490125.aspx

IO_RESOURCE_DESCRIPTOR
http://msdn.microsoft.com/en-us/library/aa491660.aspx

IoGetDeviceProperty
http://msdn.microsoft.com/en-us/library/ms801223.aspx

IRP_MN_QUERY_BUS_INFORMATION
http://msdn.microsoft.com/en-us/library/ms806439.aspx

Kernel-Mode Driver Framework
http://msdn.microsoft.com/en-us/library/aa973499.aspx

Kernel-Mode Driver Framework Extensions (Wdfkd.dll)
http://msdn.microsoft.com/en-us/library/cc267255.aspx

KMDF Samples
http://msdn.microsoft.com/en-us/library/cc463038.aspx

Plug and Play Driver Test
http://msdn.microsoft.com/en-us/library/aa906511.aspx

Tools for Verifying Drivers
http://msdn.microsoft.com/en-us/library/aa469205.aspx

Using the System-Supplied Multifunction Bus Driver
http://msdn.microsoft.com/en-us/library/ms794943.aspx

WdfDeviceCreateSymbolicLink
http://msdn.microsoft.com/en-us/library/aa491153.aspx

MSDN

Locale Identifier Constants and Strings
http://msdn.microsoft.com/en-us/library/ms776260.aspx

MSDN Blog

"How to Share Device Resources with another driver not in the same PnP Hierarchy" by Doron Holan
http://blogs.msdn.com/doronh/archive/2007/10/24/how-to-share-hw-resources-with-another-driver-not-in-the-same-pnp-hierarchy.aspx

KB Article

You cannot install a device that requires the Mf.sys device driver in Windows Vista
http://support.microsoft.com/kb/926171