Kernel-Mode Drivers: Windows 2000 DDK

Chapter 1 Supporting USB Devices

The Windows® Driver Model (WDM) supports Universal Serial Bus (USB) devices. This chapter introduces the basics of how to support a USB device on a WDM platform. It is not a general introduction to USB. Driver writers should consult the USB specification for a description of the inner workings of USB.

Built on Wednesday, June 28, 2000

Kernel-Mode Drivers: Windows 2000 DDK

1.1 USB Driver Stack

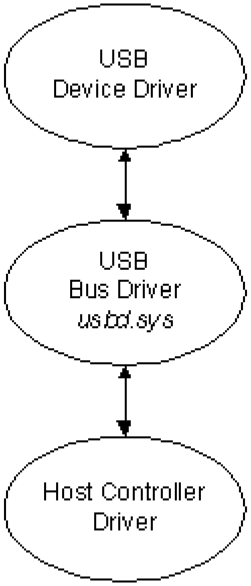

Figure 1.1 USB Driver Stack

The hub driver, *usbhub.sys*, is the device driver for each hub on the system. The bus driver, *usbd.sys,* handles all hardware-independent aspects of the USB bus, and the host controller driver handles the platform-specific interface between the system and the USB bus. A particular device is supported by a *USB client driver*.

Client drivers issue requests to their devices through a variable-length data structure known as a USB Request Block (URB). Each URB is submitted for processing to the USB bus driver by sending an IRP_MJ_INTERNAL_DEVICE_CONTROL IRP down the USB driver stack, with control code IOCTL_INTERNAL_USB_SUBMIT_URB. See IOCTL_INTERNAL_USB_SUBMIT_URB for details.

Each URB begins with a standard header. The **Length** member of the header specifies the size in bytes of the URB. The **Function** member, which must be one of the URB_FUNCTION_XXX constants, determines the type of operation requested. The bus driver uses the **Status** member to return a USB-specific status code.

The operating system provides several convenience routines for building URBs, which are documented below. To build the IOCTL_INTERNAL_USB_SUBMIT_URB request, drivers can use the routine **IoBuildDeviceIoControlRequest**.

For certain USB-defined classes of devices, Microsoft provides a class driver that serves as a client driver for all devices in that class. The operating system automatically supports devices in these classes, with no additional driver needed.

Currently, Microsoft provides class drivers for the following classes of USB devices:

- Human Interface Device (HID) devices
- USB audio

Hardware vendors do **not** write drivers for these classes of device.

Built on Wednesday, June 28, 2000

Kernel-Mode Drivers: Windows 2000 DDK

Chapter 2 Configuring USB Devices

A Universal Serial Bus (USB) device must be explicitly configured before it can handle I/O. Since a USB device may support multiple configurations, the client driver reads the devices configuration information, parses it, and chooses the appropriate configuration.

USB devices report their configuration information through descriptors. *Device descriptors* contain information on the device as a whole. *Configuration descriptors* contain a description of each individual device configuration. *String descriptors* contain Unicode text strings.

Drivers follow these steps to configure their device:

1. Read the device descriptor to determine the number of configurations the device supports. See USB Device Descriptors for details.
2. Request each configuration descriptor from the device.
3. Choose which configuration to enable. If the driver supports alternate settings for any interfaces of the configuration the driver also chooses which alternate interface to enable.

Built on Wednesday, June 28, 2000

Kernel-Mode Drivers: Windows 2000 DDK

2.1 USB Descriptors

USB supports three top-level descriptors to describe a device. See the following sections for more information:

USB Device Descriptors

USB Configuration Descriptors

USB String Descriptors

Built on Wednesday, June 28, 2000

Kernel-Mode Drivers: Windows 2000 DDK

2.1.1 USB Device Descriptors

The device descriptor contains information about a USB device as a whole. To obtain the device descriptor, use **UsbBuildGetDescriptorRequest** to build the USB request block (URB) for the request.

For example, the call

```
UsbBuildGetDescriptorRequest(
    pURB, // points to the URB to be filled in
    sizeof(_URB_CONTROL_DESCRIPTOR_REQUEST),
    USB_DEVICE_DESCRIPTOR_TYPE,
    0, // this parameter not used for device descriptors
    0, // this parameter not used for device descriptors
    pDescriptor, // points to a USB_DEVICE_DESCRIPTOR
    NULL,
    sizeof(USB_DEVICE_DESCRIPTOR),
    NULL
);
```

fills in the buffer at **pURB** with the appropriate URB. See IOCTL_INTERNAL_USB_SUBMIT_URB for a description of how to submit the URB to the bus driver. Once the bus driver completes the IRP for the request, it returns a USB_DEVICE_DESCRIPTOR in the buffer beginning at **pDescriptor**.

To determine the number of configurations a device supports, check the **bNumConfigurations** member of the returned structure.

Built on Wednesday, June 28, 2000

Kernel-Mode Drivers: Windows 2000 DDK

2.1.2 USB Configuration Descriptors

A configuration of USB device consists of a series of *interfaces*. Each interface consists of one or more *alternate settings*, and each alternate setting is made up of a set of *endpoints*.

An interface, or any of its alternate settings, specifies a class code, subclass, and protocol. Each endpoint of an interface describes a single stream of input or output for the device.

A device that supports the input or output of several different kinds of data has multiple interfaces. A device that supports several streams of the same kind of data, supports multiple endpoints on a single interface.

To obtain all this information from the device, drivers request the device's configuration descriptor. With the device returns the configuration descriptor, it also returns an *interface descriptor* for each interface or alternate setting, and an *endpoint descriptor* for each endpoint.

The driver can collect this information in two steps. The driver can issue the request to get the configuration descriptor, and pass a buffer big enough to hold the configuration descriptor alone. From the configuration descriptor, the driver can determine what size buffer is needed to hold all of the configuration information. The driver then issues the same request with the bigger buffer.

The **UsbBuildGetDescriptorRequest** routine builds the necessary request. The bus driver returns the configuration descriptor in a USB_CONFIGURATION_DESCRIPTOR structure. This structure specifies the total length of configuration information in its **wTotalLength** member.

A device may support multiple configurations, numbered starting at zero. It reports the number of configurations it supports in its device descriptor. See USB Device Descriptors for details.

The following code demonstrates how to request the configuration information for the *i*-th configuration:

```
USB_CONFIGURATION_DESCRIPTOR UCD, *pFullUCD;
UsbBuildGetDescriptorRequest(
    pURB, // points to the URB to be filled in
    sizeof(_URB_CONTROL_DESCRIPTOR_REQUEST),
    USB_CONFIGURATION_DESCRIPTOR_TYPE,
    i, // number of configuration descriptor
    0, // this parameter not used for configuration descriptors
    &UCD, // points to a USB_CONFIGURATION_DESCRIPTOR
    NULL,
    sizeof(USB_DEVICE_DESCRIPTOR),
    NULL
);
```

```
pFullUCD = ExAllocatePool(NonPagedPool, UCD.wTotalLength);
UsbBuildGetDescriptorRequest(
    pURB, // points to the URB to be filled in
    sizeof(_URB_CONTROL_DESCRIPTOR_REQUEST),
    USB_CONFIGURATION_DESCRIPTOR_TYPE,
    i, // number of configuration descriptor
    0, // this parameter not used for configuration descriptors
    pFullUCD, // points to a USB_CONFIGURATION_DESCRIPTOR
    NULL,
    UCD.wTotalLength,
    NULL
);
```

Each interface descriptor is stored in a USB_INTERFACE_DESCRIPTOR structure. The driver returns one interface descriptor for each interface or alternate setting. The zero-based **bInterfaceNumber** member of USB_INTERFACE_DESCRIPTOR distinguishes interfaces within a configuration. For a given interface the zero-based **bAlternateSetting** member distinguishes between alternate settings of the interface. Each endpoint descriptor is stored in a USB_ENDPOINT_DESCRIPTOR structure.

When a device reports a configuration, each interface descriptor is followed in memory by all of the endpoint descriptors for the interface and alternate setting. The device returns interface descriptors in order of **bInterfaceNumber** values and then in order of **bAlternateSetting** values.

For example, consider a device that supports a configuration with two interfaces, and the first interface supports two alternate settings. The configuration information is laid out in memory as follows:

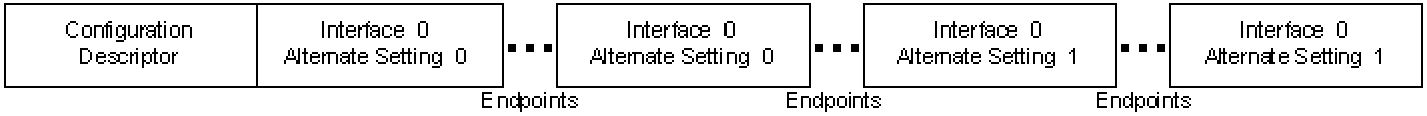
Configuration Descriptor
Interface 0 Alternate Setting 0
Interface 0 Alternate Setting 0
Interface 0 Alternate Setting 1
Interface 0 Alternate Setting 1
Endpoints
Endpoints
Endpoints

Figure 2.1 Example of Configuration Descriptor Layout

The operating system provides a helper function, **USBD_ParseConfigurationDescriptorEx**, to search for a given interface descriptor within the configuration. The driver provides a starting position within the configuration, and optionally an interface number, an alternate setting, a class, a subclass, or a protocol. The routine returns a pointer to the next matching interface descriptor.

To examine a configuration descriptor for an endpoint or string descriptor, use the **USBD_ParseDescriptors** routine. The caller provides a starting position within the configuration and a descriptor type, such as USB_STRING_DESCRIPTOR_TYPE or USB_ENDPOINT_DESCRIPTOR_TYPE. The routine returns a pointer to the next matching descriptor.

Built on Wednesday, June 28, 2000

Kernel-Mode Drivers: Windows 2000 DDK

2.1.3 USB String Descriptors

Device, configuration, and interface descriptors may contain references to string descriptors. String descriptors are referenced by their one-based index number. A string descriptor contains one or more Unicode strings; each string is a translation of the others into another language.

Drivers use **UsbBuildGetDescriptorRequest**, with *DescriptorType* = USB_STRING_DESCRIPTOR_TYPE, to build the request to fetch a string descriptor. The *Index* parameter specifies the index number, and the *LanguageID* parameter specifies the language ID (the same values are used as in Microsoft® Win32® LANGID values). Drivers can request the special index number of zero to determine which language IDs the device supports. For this special value, the device returns an array of language IDs rather than a Unicode string.

Since the string descriptor consists of variable-length data, the driver must fetch it in two steps. First the driver must issue the request, passing a data buffer large enough to hold the header for a string descriptor, a USB_STRING_DESCRIPTOR structure. The **bLength** member of USB_STRING_DESCRIPTOR specifies the size in bytes of the entire descriptor. The driver then makes the same request with a data buffer of size **bLength**.

The following code demonstrates how to request the *i*-th string descriptor, with language ID *langID*:

```
USB_STRING_DESCRIPTOR UCD, *pFullUSD;
UsbBuildGetDescriptorRequest(
    pURB, // points to the URB to be filled in
    sizeof(_URB_CONTROL_DESCRIPTOR_REQUEST),
    USB_STRING_DESCRIPTOR_TYPE,
    i, // index of string descriptor
    langID, // language ID of string.
    &USD, // points to a USB_STRING_DESCRIPTOR.
    NULL,
    sizeof(USB_STRING_DESCRIPTOR),
    NULL
);
pFullUSD = ExAllocatePool(NonPagedPool, UCD.wTotalLength);
UsbBuildGetDescriptorRequest(
    pURB, // points to the URB to be filled in
    sizeof(_URB_CONTROL_DESCRIPTOR_REQUEST),
    USB_STRING_DESCRIPTOR_TYPE,
    i, // index of string descriptor
    langID, // language ID of string
    pFullUSD,
    NULL,
    USD.bLength,
    NULL
);
```

Built on Wednesday, June 28, 2000

Kernel-Mode Drivers: Windows 2000 DDK

2.2 Selecting the USB Device Configuration

To configure a USB device, the driver chooses which configuration to enable, and within a configuration, which alternate settings of each interface to enable. The driver packages these choices in an URB_FUNCTION_SELECT_CONFIGURATION URB, which it sends to the USB bus driver. The bus driver enables each interface, and sets up a communication channel to each endpoint within the interface. In USB terminology, this connection is known as a *pipe*. The bus driver returns a handle for the configuration, for each interface, and for each pipe. The device driver uses the configuration handle to change configuration settings, and uses the pipe handles to perform reads and writes on a pipe.

Drivers use the **USBD_CreateConfigurationRequestEx** routine to build the URB_FUNCTION_SELECT_CONFIGURATION URB. A driver passes an array of USBD_INTERFACE_INFORMATION structures to the routine, which specifies the alternate setting of each interface to enable. The *InterfaceDescriptor* member of each structure points to the interface descriptor for the alternate setting. Each USBD_INTERFACE_INFORMATION entry contains within it, in its **Pipes** member, an array USBD_PIPE_INFORMATION structures — one for each endpoint of that particular interface and alternate setting.

To populate the *InterfaceDescriptor* members of the array, drivers can use the **USBD_ParseConfigurationDescriptor** support routine. For example, the following code demonstrates how a driver may use the routine to fill the array with pointers to alternate setting zero of each interface.

```
PUSB_CONFIGURATION_DESCRIPTOR pConfigurationDescriptor;
/* pConfigurationDescriptor points to the descriptor previously
   requested from the driver. */

PUSBD_INTERFACE_INFORMATION pInterfaceInfo;
/* pInterfaceInfo points to an array with
   pConfigurationDescriptor->bNumInterfaces entries. */

PUSB_INTERFACE_DESCRIPTOR pCurrentDescriptor;
LONG InterfaceNumber = 0;

PURB pUrb;

pCurrentDescriptor = pConfigurationDescriptor;

for (InterfaceNumber = 0;
     InterfaceNumber < pConfigurationDescriptor->bNumInterfaces;
     InterfaceNumber++
    )
{
    /* Get the next matching descriptor. Here we implicitly use
       the fact that the interface descriptors are laid out in
       order in memory.
    */
    pCurrentDescriptor = USBD_ParseConfigurationDescriptor(
                            pConfigurationDescriptor,
                            pCurrentDescriptor,
                            InterfaceNumber, // interface number
                            0, // alternate setting
                            -1,
                            -1,
                            -1,
                         );
    pInterfaceInfo[InterfaceNumber] = pCurrentDescriptor;
}
// allocate the URB
pUrb = USBD_CreateConfigurationRequestEx(
           pConfigurationDescriptor,
           pInterfaceInfo
       );
```

The **USBD_CreateConfigurationRequestEx** routine allocates the URB for the driver. Within the URB, the routine allocates an USBD_INTERFACE_INFORMATION structure for each interface descriptor the driver specifies. The routine initializes the **InterfaceNumber**, **AlternateSetting**, **NumberOfPipes**, **Pipes [i].MaximumTransferSize**, **and Pipes[i].PipeFlags** members.

USBD_CreateConfigurationRequestEx initializes **Pipes[i].MaximumTransferSize** to the default maximum transfer size for a single URB read/write request. The driver can specify a different maximum transfer size in the **Pipes[i].MaximumTransferSize**, if it also specifies the USBD_PF_CHANGE_MAX_PACKET flag in **Pipes [i].PipeFlags**.

Once the bus driver receives the URB, it fills in the rest of the members of each USBD_INTERFACE_INFORMATION structure. In particular, the **Pipes** array member contains information on each pipe created. For example, the **Pipes[i].PipeHandle** member contains a handle the driver uses to send read/write requests to the pipe. The **Pipes[i].PipeType** member specifies the type of transfers the pipe supports.

Within the **UrbSelectConfiguration** member of the URB, the bus driver returns a handle the driver can use later to select alternate interface settings. To select an alternate setting for an interface, the driver submits an URB_FUNCTION_SELECT_INTERFACE URB. The driver can use the **UsbBuildSelectInterfaceRequest** routine to build this URB.

To disable the device, the driver creates and submits an URB_FUNCTION_SELECT_CONFIGURATION request with a NULL configuration descriptor. The following code shows how to use the **USBD_CreateConfigurationRequestEx** routine to allocate the proper URB.

```
URB Urb;
UsbBuildSelectConfigurationRequest(
  &Urb,
  sizeof(_URB_SELECT_CONFIGURATION),
  NULL
);
```

The URB_FUNCTION_SELECT_CONFIGURATION and URB_FUNCTION_SELECT_INTERFACE requests can fail if there is insufficient bandwidth to support the isochronous, control, and interrupt endpoints within enabled interfaces. When that happens, the bus driver sets the Status member of the URB header to USBD_STATUS_NO_BANDWIDTH.

Built on Wednesday, June 28, 2000

Kernel-Mode Drivers: Windows 2000 DDK

Chapter 3 USB Device I/O

When the bus driver configures the device, it sets up a pipe to device endpoints. The bus driver returns a USBD_PIPE_INFORMATION structure for the pipe, which contains information about the pipe created. The **PipeHandle** member contains a pipe handle to use for I/O transactions on that pipe. See USB Configuration Descriptors for a description of Universal Serial Bus (USB) device configuration and obtaining the pipe handles.

USB devices can support different types of pipes. Drivers can determine the pipe type by examining the **PipeType** member of USBD_PIPE_INFORMATION. The different pipe types require different types of USB request blocks (URB) to perform I/O transactions, but each such URB shares certain members: **TransferFlags**, **TransferBuffer**, **TransferBufferLength**, **TransferBufferMDL**.

The **TransferFlags** member specifies the direction of data transfer. The driver specifies no flag to send data to the device. To read data from the device, the driver specifies the USBD_TRANSFER_DIRECTION_IN flag. By default, the host controller driver signals an error to the device if it receives a packet smaller than the maximum packet size for the endpoint. To override this behavior, the driver specifies the USBD_SHORT_TRANSFER_OK flag as well. Specific URB types may specify additional flags.

The data may be read from or written to either a buffer resident in memory or an MDL. In either case, the driver specifies the size of the buffer in the **TransferBufferLength** member. The driver provides a resident buffer in the **TransferBuffer** member and an MDL in the **TransferBufferMDL** member. Whichever one the driver provides, the other must be NULL.

If an I/O request on a control, interrupt, or bulk pipe fails, the pipe stalls. The driver checks for this condition by testing the value of the USB_HALTED macro on the **Hdr.Status** member of the returned URB.

If the pipe is stalled, the driver must reset the pipe before any other I/O requests on the pipe can be handled. The driver does this by submitting a URB_FUNCTION_RESET_PIPE request, with the **UrbPipeRequest.PipeHandle** member set to the pipe handle of the stalled pipe. The driver can abort all I/O requests on the pipe with the URB_FUNCTION_ABORT_PIPE request.

Specifics about each pipe type is provided in the following sections:

USB Bulk and Interrupt Transfer

USB Isochronous Transfer

USB Control Transfer

Built on Wednesday, June 28, 2000

Kernel-Mode Drivers: Windows 2000 DDK

3.1 USB Bulk and Interrupt Transfer

USB devices support bulk transfer pipes for data that does not require delivery in a guaranteed amount of time. The USB host controller gives a lower priority to bulk transfer than the other types of transfer.

USB devices use interrupt transfer for small packets of data that arrive asynchronously. Since all I/O is under direct control of the driver, the driver must poll the device at the frequency specified in the **Interval** member of the USBD_PIPE_INFORMATION structure for the pipe.

Drivers use the **UsbBuildInterruptOrBulkTransferRequest** routine to build the URB for I/O transactions on bulk or interrupt pipes. See USB Device I/O for a description of the information the driver provides for I/O transactions.

Built on Wednesday, June 28, 2000

Kernel-Mode Drivers: Windows 2000 DDK

3.2 USB Isochronous Transfer

USB devices support isochronous transfer pipes for time-dependent data that does not require guaranteed delivery. The driver submits the URB_FUNCTION_ISOCH_TRANSFER URB to perform I/O on the pipe.

Isochronous transfer is packet-based — the device reads or writes a single packet of data each millisecond frame. Fortunately, the driver can send several consecutive packets with a single URB.

The size of the URB_FUNCTION_ISOCH_TRANSFER URB varies with the number of packets the driver sends. Drivers can use the GET_ISO_URB_SIZE macro to determine the number of bytes to allocate for the URB. The **UrbIsochronousTransfer.IsoPacket** array member of the URB describes the length and offset of each packet within the buffer specified in the **UrbIsochronousTransfer.TransferBuffer** or **UrbIsochronousTransfer.TransferBufferMDL** members.

Here is an example of how to create an URB for a data buffer that can hold ten packets, each sixteen bytes long:

```
#define NUMBER_OF_PACKETS 10
#define PACKET_SIZE 16

ULONG UrbSize;
PURB pUrb;

UrbSize = GET_ISO_URB_SIZE(NUMBER_OF_PACKETS);
pUrb = (PURB) ExAllocatePool(NonPagedPool, UrbSize);
urb->UrbIsochronousTransfer.Hdr.Length = size;
urb->UrbIsochronousTransfer.Hdr.Function = URB_FUNCTION_ISOCH_TRANSFER;
urb->PipeHandle = PipeHandle;
        .
        .
        .
urb->UrbIsochronousTransfer.NumberOfPackets = NUMBER_OF_PACKETS;

ULONG i;
for (i = 0; i<NUMBER_OF_PACKETS; i++) {
  urb->UrbIsochronousTransfer.IsoPacket[i].Offset = i*PACKET_SIZE;
  urb->UrbIsochronousTransfer.IsoPacket[i].Length = PACKET_SIZE;
}
```

The **UrbIsochronousTransfer.StartFrame** member of the URB specifies the starting USB frame number for the transaction. The driver can use the URB_FUNCTION_GET_CURRENT_FRAME_NUMBER URB to request current frame number.

For the first transaction on the pipe after being opened or reset, the driver can specify the USB_START_ISO_TRANSFER_ASAP flag in **UrbIsochronousTransfer.TransferFlags**. The bus driver will begin the transaction in the next available frame.

Built on Wednesday, June 28, 2000

Kernel-Mode Drivers: Windows 2000 DDK

3.3 USB Control Transfer

All USB devices support endpoint zero for standard control requests. Devices can support additional endpoints for custom control requests.

For endpoints other than endpoint zero, drivers issue the URB_FUNCTION_CONTROL_TRANSFER URB request. The **UrbControlTransfer.SetupPacket** member of the URB specifies the initial setup packet for the control request. See the USB specification for the place of this packet in the protocol.

The USB driver stack supports the endpoint zero control requests as follows:

USB Feature Requests

USB devices support feature requests to enable or disable certain Boolean device settings. Drivers use the **UsbBuildFeatureRequest** support routine to build the URB feature request.

USB Status Requests

USB devices support status requests to get or set the USB-defined status bits of a device, endpoint, or interface. Drivers use the **UsbBuildGetStatusRequest** to build the URB status request.

Getting or Setting the USB Configuration

See Configuring USB Devices for a description of how to set the USB configuration.

Drivers use the URB_FUNCTION_GET_CONFIGURATION URB to request the current configuration. The driver passes a one-byte buffer in **UrbControlGetConfiguration.TransferBuffer**, which the bus driver fills in with the current configuration number.

Getting USB Descriptors

See USB Descriptors for a description of how to get USB descriptors.

Getting or Setting USB Interfaces

To select an alternate setting for an interface, the driver submits an URB_FUNCTION_SELECT_INTERFACE URB. The driver can use the **UsbBuildSelectInterfaceRequest** routine to format this URB. The caller supplies the handle for the current configuration, the interface members, and the new alternate settings.

Drivers use the URB_FUNCTION_GET_CONFIGURATION URB to request the current setting of an interface. The **UrbControlGetInterface.Interface** member of the URB specifies the interface number to query. The driver passes a one-byte buffer in **UrbControlGetInterface.TransferBuffer**, which the bus driver fills in with the current alternate setting.

USB Class and Vendor Requests

To submit USB class control requests and vendor endpoint zero control requests, drivers use one of the URB_FUNCTION_CLASS_*XXX* or URB_FUNCTION_VENDOR_*XXX* requests. Drivers can use the **UsbBuildVendorRequest** routine to format the URB.

Built on Wednesday, June 28, 2000