USB Composite Device

1. What is composite device

Composite device is defined in the USB spec as follows (usb_20.pdf 5.2.3),

"A device that has multiple interfaces controlled independently of each other is referred to as a composite device."

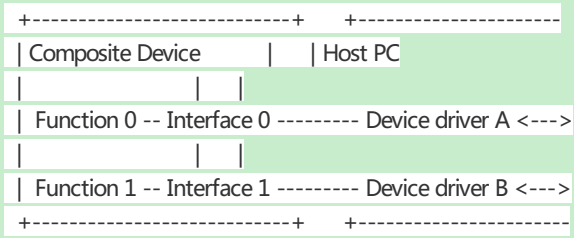
Using composite device, multiple functions are combined into a single device.

Ex.

- Keyboard + Mouse
- Video + USB Hard disk
- I/O device (HID + USB_bulk)

Another advantage of composite device is that it eases the device driver development.

OS assigns a separate device driver to each Interface of the composite device as follows. Therefore, a dedicated monolithic driver is not required for newly designed device; you can realize it using existing drivers.



When OS has some required drivers as built-in, they are available for the composite device. These OS built-in device drivers are called as USB class driver.

"Approved Class Specification Documents" from USB.org

http://www.usb.org/developers/devclass_docs#approved

Windows have these built-in class drivers.

"USB FAQ: Introductory Level - USB Class Drivers" from MS WHDC

http://www.microsoft.com/whdc/system/bus/usb/USBFAQ_intro.mspx

Please note, available drivers for a composite device are not limited only to class drivers. Any driver can be applied, as long as it doesn't require a device class (class defined in device descriptor). For example, SiLabs USB_INT and USB_bulk drivers are also applicable for composite devices.

2. How Windows handles a composite device

When a device satisfies these three requirement, Windows system recognizes the device as a composite device.

1. The class field of the device descriptor equals to zero: (bDeviceClass) = (0x00)
2. Single Configuration
3. Multiple Interfaces

[Note]
WinXP SP2, Win2k3 SP1 and Vista supports this alternative requirement.
1'. The class, subclass and protocol fields of the device are that of Interface Association Descriptor:
(bDeviceClass, bDeviceSubClass, bDeviceProtocol) = (0xEF, 0x02, 0x01)

When an USB device is plugged in to a PC, the system reads out the device descriptor of the device and makes these Device ID.

USB\VID_vvvv&PID_pppp
USB\VID_vvvv&PID_pppp&REV_rrrr
(vvvv, pppp, rrrr: four digit hex number for the VID, PID, device release number. Matches to idVendor/ idProduct/ bcdDevice, defined in the device descriptor)

The system searches device database on the registry, installed by INF files. When the system finds the Device ID in a device record, it assigns the device driver specified by the record. However, when it cannot find any matched record, and the device Configuration satisfies above criteria, the generic composite parent driver is loaded. This parent driver parses the Configuration of the device, and assigns this Device ID to each Interfaces.

USB\VID_vvvv&PID_pppp&MI_mm

USB\VID_vvvv&PID_pppp&REV_rrrr&MI_mm
(mm: Interface number of the corresponding function, two digit hex number)

As of the Interface which specifies a class, the system also assigns this Compatible ID.

USB\CLASS_cc
USB\CLASS_cc&SUBCLASS_ss
USB\CLASS_cc&SUBCLASS_ss&PROT_pp
(cc/ ss / pp: two digits hex number.
bInterfaceClass/ bInterfaceSubClass/ bInterfaceProtocol, from the Interface descriptor)

The system searches these Device ID and Compatible ID in the device database again. When it finds a matched record, it assigns the specified device driver to the Interface. However, when it cannot find any matched record, it shows 'New Hardware Wizard' to users and asks them to install the device driver.
"Enumeration of the Composite Parent Device" from MSDN
http://msdn2.microsoft.com/en-us/library/aa476434.aspx

3. Implementation

On USB application, firmware, device driver and host application are closely related. It is desirable to make separate prototypes of firmwares and host applications for each Interface first. After confirming them to work properly, combine them together.

3.1 Device firmware

When the source code is well organized, modification of the firmware is easy. Based on one of the prototypes, copy a part of code from other prototypes and insert it to corresponding part of the base source code.
In other word, the source codes should be organized considering to make it a composite device. When these parameters are defined by #define macro or enum instead of direct number in each prototype, the combination of prototypes is done smoothly.

- Interface number
- Endpoint address
- Endpoint status (IDLE / HALT)

3.1.1 Descriptors

3.1.1.1 Device descriptor

- bDeviceClass: Must be assigned to zero
- idVendor, idProduct: VID/PID must be unique, to avoid conflict to other devices.

3.1.1.2 Configuration descriptor

- wTotalLength: The total number of bytes of Configuration set, includes all of Interface sets
- bNumInterfaces: Number of Interfaces included in this Configuration

Configuration set means these descriptors, for example.

Configuration descriptor
- Interface descriptor (0)
- - accessory descriptor, such as HID class descriptor, if any
- - Endpoint descriptors
- Interface descriptor (1)
- - accessory descriptor, such as HID class descriptor, if any
- - Endpoint descriptors

HID report descriptor and String descriptors are not included in the Configuration set.

3.1.1.3 Interface descriptors

- bInterfaceNumber: Index of Interfaces, starting from zero
- bInterfaceClass: Specify class code for this Interface

If any specific class code is not assigned to the Interface, set bInterfaceClass to 0xFF (Vendor specific). 0x00 (Reserved) would work, but 0xFF is better.

3.1.1.4 Endpoint descriptors

- bEndpointAddress: must be unique on each Endpoint descriptor

Any duplicated Endpoint across Interfaces is not allowed in a composite device.

3.1.2 Standard requests

3.1.2.1 Additional Descriptor handling
Get_Descriptor must support additional descriptors asked by the host.

- Configuration descriptor: return full Configuration set (see 3.1.1.2)
- Class-specific descriptors:

When the descriptor type in request (MSB of wValue) is not Device(1), Configuration(2) or String(3), the request may be for class-specific descriptor (in full-speed devices). In this case, wIndex field of the Setup data shows the Interface number in question. According to the class specified by the Interface (wIndex), Get_Descriptor must return the class-specific descriptor specified by the MSB of wValue.
When the class supports Set_Descriptor request, it must be handled similarly to Get_Descriptor.
Interface and Endpoint descriptors cannot be directly accessed with Get_Descriptor or Set_Descriptor. Therefore, Get_Descriptor and Set_Descriptor have no need to support them.
3.1.2.2 Additional Interfaces handling
wIndex field of the Setup data shows the Interface number in question. When this Interface number matches to the additional Interfaces, handle the requests as follows.

- Get_Status: return Zero
- Get_Interface: return current alternate Interface number
- Set_Interface: set current alternate Interface to one specified by the request

When the Interface in question doesn't have any alternate Interface, Get_Interface returns Zero. And Set_Interface succeeds when the wValue is Zero, otherwise return STALL.
3.1.2.3 Additional Endpoints handling

- Get_Status: return HALT condition of the Endpoint
- Clear_Feature: recover the Endpoint from HALT
- Set_Feature: set the Endpoint to HALT
- Set_Configuration: Setup additional Endpoints

When Get_Status, Clear_Feature and Set_Feature are issued to an Endpoint, wIndex field of the Setup data indicates the Endpoint address.
When the Interfaces doesn't have any alternate Interface, set up the Endpoints in Set_Configuration request. As of the Interface with any alternate Interface, set up the Endpoints belonging to the Interface in Set_Interface.

3.1.3 Class-specific requests
wIndex field of the Setup data of the request indicates the Interface to which this request is issued. Therefore, dispatch the request by wIndex first, and copy the each handler for class-specific requests from the prototypes under each case.

3.1.4 Endpoint handling
When the Endpoint address and Endpoint status are defined by macro, modification on this part finishes by copying the Endpoint handler of each prototype to the base one.

3.2 Device driver and host application
OS built-in class drivers are designed to work with composite devices. In most case, these drivers are applicable to composite devices without any change of default INF file. However, device drivers provided by vendors are not always designed to work with composite devices. INF file and host application code should be modified for these drivers. The device driver itself should work without any change. Of course, rare exception may exist.
3.2.1 INF file
When the device driver requires an INF file even for single use, the INF file is required as a part of composite device. The INF file defines the Device ID as follows.
USB\VID_vvvv&PID_pppp

USB\VID_vvvv&PID_pppp&REV_rrrr

For a composite device, the Interface number must be added to this Device ID.

USB\VID_vvvv&PID_pppp&MI_mm

USB\VID_vvvv&PID_pppp&REV_rrrr&MI_mm

For example, when you add the USB_Bulk function (Interface with bulk IN/OUT Endpoints) to your composite device as the Interface number 1, the Device ID in the INF file (SilabsBulk.inf) is modified as follows.

USB\VID_vvvv&PID_pppp&MI_01

(vvvv, pppp: VID/PID must be unique)

3.2.2 Endpoint address and pipe name / device path name

When two or more devices are combined into a single composite device, the Endpoint addresses must be often changed to fit to the newly designed device. Usually, the pipe name and device path name from device drivers are designed to hide the Endpoint address. Therefore, in most case, Endpoint address reassignment doesn't affect to the host application.
However, there are some drivers which expose the Endpoint address directly to the pipe name. For these drivers, the host application must be modified to reflect the Endpoint address assignment. Confirmation for this point is desirable before planning a new device.

- OS built-in class drivers hide Endpoint address behind its device path name.
- MS WinDDK bulkusb and isousb example driver hide Endpoint address behind the pipe name (a part of device path name).
- SiLabs USB_INT and USB_Bulk device drivers hide it behind the pipe name (a part of device path name).
- Cypress ezusb.sys driver hides Endpoint address behind its pipe number.
- Cypress CyUSB.sys driver exposes Endpoint address directly. But when the code follows the example of CCyUSBDevice::BulkInEndPt in CyAPI.chm, the Endpoint address is hidden behind the index of the Endpoint array.

When a device applies the same class to multiple Interfaces, the host application should be modified to distinguish these Interfaces.

Enumeration of USB Composite Devices

When a new USB device is connected to a host machine, the USB bus driver creates a physical device object (PDO) for the device and generates a PnP event to report the new PDO. The operating system then queries the bus driver for the hardware IDs associated with the PDO.
For all USB devices, the USB bus driver reports a *device ID* with the following format:
`USB\VID_xxxx&PID_yyyy`
Note *xxxx* and *yyyy* are taken directly from **idVendor** and **idProduct** fields of the device descriptor, respectively.
The bus driver also reports a compatible identifier (ID) of `USB\COMPOSITE`, if the device meets the following requirements:

- The device class field of the device descriptor (**bDeviceClass**) must contain a value of zero, or the class (**bDeviceClass**), subclass (**bDeviceSubClass**), and protocol (**bDeviceProtocol**) fields of the device descriptor must have the values 0xEF, 0x02 and 0x01 respectively, as explained in USB Interface Association Descriptor.
- The device must have multiple interfaces.
- The device must have a single configuration.

The bus driver also checks the device class (**bDeviceClass**), subclass (**bDeviceSubClass**), and protocol (**bDeviceProtocol**) fields of the device descriptor. If these fields are zero, the device is a composite device, and the bus driver reports an extra compatible identifier (ID) of USB\COMPOSITE for the PDO.
After retrieving the hardware and compatible IDs for the new PDO, the operating system searches the INF files. If one of the INF files contains a match for the device ID, Windows loads the driver that is indicated by that INF file and the generic parent driver does not come into play. If no INF file contains the device ID, and the PDO has a compatible ID, Windows searches for the compatible ID. This produces a match in Usb.inf and causes the operating system to load the USB Generic Parent Driver (Usbccgp.sys).
If you want the generic parent driver to manage your device, but your device does not have the characteristics necessary to ensure

that the system will generate a compatible ID of USB\COMPOSITE, you will have to provide an INF file that loads the generic parent driver. The INF file should contain a needs/includes section that references Usb.inf.

If your composite device has multiple configurations, the INF file you provide must specify which configuration the generic parent should use in the registry. The necessary registry keys are described in Selecting the Configuration for a Composite USB Device.

Related Topics

USB Generic Parent Driver (Usbccgp.sys)

System-Supplied USB Drivers

Enumeration of Interfaces on USB Composite Devices

Interfaces on a composite USB device can be grouped in collections or represent one USB function individually. When the interfaces are not grouped in collections, the generic parent driver creates a PDO for each interface and generates a set of hardware IDs for each PDO.

The *device ID* for an interface PDO has the following form:

```
USB\VID_v(4)&PID_p(4)&MI_z(2)
```

In these IDs:

- *v(4)* is the four-digit vendor code that the USB standards committee assigns to the vendor.
- *p(4)* is the four-digit product code that the vendor assigns to the device.
- *z(2)* is the interface number that is extracted from the **bInterfaceNumber** field of the interface descriptor.

The generic parent driver also generates the following compatible IDs by using the information from the interface descriptor (**USB_INTERFACE_DESCRIPTOR**):

```
USB\CLASS_d(2)&SUBCLASS_s(2)&PROT_p(2)
USB\CLASS_d(2)&SUBCLASS_s(2)
USB\CLASS_d(2)
```

In these IDs:

- *d(2)* is the class code (**bInterfaceClass**)
- *s(2)* is the subclass code (**bInterfaceSubClass**)
- *p(2)* is the protocol code (**bInterfaceProtocol**)

Each of these codes is a four-digit number.

Related topics

Enumeration of Interface Collections on USB Composite Devices

USB Generic Parent Driver (Usbccgp.sys)

System-Supplied USB Drivers

USB Generic Parent Driver (Usbccgp.sys)

This section describes the Usbccgp.sys driver provided by Microsoft for composite devices.

Many USB devices expose multiple *USB interfaces*. In USB terminology, these devices are called *composite devices*. Microsoft Windows 2000 and Windows 98 operating systems include a generic parent facility in the USB bus driver (Usbhub.sys) that exposes each interface of the composite device as a separate device. In Microsoft Windows XP and Windows Me, this facility is streamlined and improved by transferring it to an independent driver called the *USB generic parent driver* (Usbccgp.sys). Using the features of the generic parent driver, device vendors can make selective use of Microsoft-supplied driver support for some interfaces.

The interfaces of some composite device operate independently. For example, a composite USB keyboard with power buttons might have one interface for the keyboard and another interface for the power buttons. The USB generic parent driver enumerates each of these interfaces as a separate device. The operating system loads the Microsoft-supplied keyboard driver to manage the keyboard interface, and the Microsoft-supplied power keys driver to manage the power keys interface.

If the composite device has an interface that is not supported by native Windows drivers, the vendor of the device should provide a driver for the interfaces and an INF file. The INF file should have an INF *DDInstall* section that matches the device ID of interface. The INF file must not match the device ID for the composite device itself, because this prevents the generic parent driver from loading. For an explanation of how the operating system loads the USB generic parent driver, see Enumeration of USB Composite Devices.

Some devices group interfaces into *interface collections* that work together to perform a particular *function*. When interfaces are grouped in interface collections, the generic parent driver treats each collection, rather than each individual interfaces, as a device. For more information on how the generic parent driver manages interface collections, see Enumeration of Interface Collections on USB Composite Devices.

After the operating system loads the client drivers for the interfaces of a composite device, the generic parent driver multiplexes the data flow from the client drivers, combining these separate interactions into a single data stream for the composite device. The generic parent is power policy owner for the entire composite device and all of its interfaces. It also manages synchronization and PnP requests.

The generic parent driver can simplify the task for vendors of composite hardware, if Microsoft-supplied drivers support some interfaces but not others. Vendors of such devices need only supply drivers for the unsupported interfaces, because the generic parent driver facilitates the use of Microsoft-supplied drivers for the supported interfaces.

The following sections describe the features and functions of the generic parent driver:

Enumeration of USB Composite Devices

Descriptors on USB Composite Devices

Enumeration of Interfaces on USB Composite Devices

Enumeration of Interface Collections on USB Composite Devices

Content Security Features in Usbccgp.sys

Related topics

System-Supplied USB Drivers