Universal Serial Bus Common Class Specification

SYSTEMSOFT® CORPORATION

INTEL® CORPORATION

Revision 1.0

December 16, 1997

Scope of this Revision

Revision 1.0 of this document includes all modifications suggested at face to face meeting in order to produce a 1.0 release candidate.

Revision History

Revision	Issue Date	Comments
1.0	December 14, 1997	Final updates as agreed by CCS CWG at December face to face meeting for 1.0.
0.9	October 31, 1997	Conversion to master document containing references to the USB Feature Specifications.
0.8c	July 14, 1997	Minimal update from RRs for DWG face-to-face. Revision marks in Dynamic Interfaces from 0.8b were cleared with the understanding that this section will continue to change.
0.8b	May 26, 1997	Updated for June DWG face to face.
0.8a	April 5, 1997	Updated for April DWG face to face.
0.8rc	January 17, 1997	Describes why class specifications are being developed for USB devices and what a specific class document should include. This document also describes attributes and services that are common to more than one class of USB device, but are not required by all USB devices.
0.7c	January 2, 1997	Updated re: feedback on shared endpoints and for greater clarity and examples. Added driver identification discussion from white paper.
0.7b	November 29, 1996	Updated per proposed resolution of review requests 48 & 49.
0.7	October 7, 1996	First round comments included
0.6	August 25, 1996	First draft to establish concept

Contributors

Gal Ashour	IBM
Paul Berg	SystemSoft Corporation
Mike Bourdess	AMD
Shelagh Callahan	Intel Corporation
Ed Endejan	U.S. Robotics
Geert Knapen	Philips
John Howard	Intel Corporation
Dave Lawrence	SystemSoft Corporation
Mark Lavelle	Logitech
Jordan Brown	Sun Microsystems
Mark Williams	Microsoft Corporation

Please send comments via electronic mail to david_g_lawrence@iname.com and John.Howard@intel.com

Table of Contents

1. Introduction

1.1 Purpose

This document describes requirements for USB Classes and their specifications. In addition, this document describes attributes and services that may be common to more than one class, but are not required for all USB devices.

1.2 Scope

The information provided in this document serves as a guideline for the development of USB class specifications, as well as defining common class capabilities. As such, it defines how devices and interfaces using the class or common capability are to be implemented and how developers of generic or adaptive device drivers will interact with compliant implementations.

1.3 Related Documents

Universal Serial Bus Specification, Revision 1.0

1.4 Terms and Abbreviations

ADAPTIVE DEVICE DRIVER	A device driver providing support by using that device's self-description.
CAPABILITY	A visible function provided by one or more interfaces, such as speakers, keypad, etc.
CLASS DRIVER	An adaptive device driver based on a class definition.
GENERIC DEVICE DRIVER	See Adaptive Device Driver.

2. Management Overview

This section is an overview of the contents of this document and provides a brief summary of each of the subsequent sections. It does not establish any requirements or guidelines.

2.1 USB Classes

A USB Class describes a group of devices or interfaces with similar attributes or services. A Class Specification defines the requirements for such a related group. A complete class specification allows manufacturers to create implementations which may be managed by an adaptive device driver. Adaptive drivers are intended to be developed by operating system and third party software vendors as well as manufacturers supporting multiple products.

2.2 Class Specification Format

This document describes a suggested format for class specifications. Since the actual definition of what constitutes a class may vary from class to another, class specification developers are not required to follow this format. This document should instead be considered a guideline. The overriding requirement is that the class specification provides sufficient information for the:

- driver developer to create an adaptive driver that is capable of operating a device or interface.

- manufacturer to build an operational device or interface.

which follows the class specification.

2.3 Common Attributes and Services

The development of class specifications made evident attributes and services which a number of classes have in common. To encourage the common definition of such attributes and services, this document contains introductions for a set of attributes and services with titles of the actual specification documents where the requirements for classes using these attributes and services are described.

It is not required that an implementation use these definitions, but if a class developer wishes to incorporate these attributes and services, the advantages of using common definitions should be strongly considered. For example, an operating system may choose to incorporate standard support for a common feature, just as the standard device framework is supported in a standard manner. Also, consider the ease with which a class driver developer might understand the attribute or service based on development of other class drivers, which use similar features. This also illustrates the ability of class driver developers to incorporate common code to handle such features across classes of devices. Eventually, following common designs may allow silicon developers to offer hardware support for some features.

3. USB Classes

3.1 What is a Class?

For the purposes of USB, a class is a group of devices (or interfaces) which have certain attributes or services in common. Typically, two devices (or interfaces) are placed in the same class if they provide or consume data streams having similar data formats or if both devices use a similar means of communicating with a host system.

USB classes are primarily used to describe the manner in which an interface communicates with the host, including both the data and control mechanisms. However, some USB classes also have the secondary purpose of identifying in whole or in part the capability provided by that interface. Thus the class information can be used to identify a driver responsible for managing the:

- interface's connectivity.

- capability provided by the interface.

3.2 Why Have Classes?

Grouping devices or interfaces together in classes and then specifying the characteristics in a Class Specification allows the development of host software which can manage multiple implementations based on that class. Such host software adapts its operation to a specific device or interface using descriptive information presented by the device. A class specification serves as a framework defining the minimum operation of all devices or interfaces which identify themselves as members of the class.

By developing in compliance with a Class Specification, entities other than the device manufacturer are able to develop software which can interact with the device. This relieves the device manufacturer from having to develop software for every combination of host platform and operating system that potentially could support the device. It also makes it easier for a device to fit into a platform/operating system's system management schemes without requiring additional support from the manufacturer. Thus, the device can be more compatible in areas such as power and connection management.

In addition, operating system vendors desiring to support a number of USB devices need to develop only a few class-specific drivers in order to make a wide-range of USB devices available for their environment. In this way, end-users have the ability to attach the latest USB devices to their system and device manufacturers get another market for their devices without requiring the development effort and distribution problems related to the use of device-specific drivers.

The process of developing a successful class specification requires operating system vendors and device manufacturers to cooperate in the definition of a class. This public review process often results in improved communications between hardware and software as both sides develop a better understanding of the requirements and constraints the other faces in supporting a particular class of device.

The ability of device manufacturers to share their experiences and perspectives improves the abstraction presented by a class specification. An appropriate level of abstraction allows a simpler interface between device and host. The class specification also identifies characteristics which might vary between implementations and establishes domains or ranges for that variation and a method for a device to communicate its particular implementation requirements.

3.3 USB Relationships

USB changes the traditional relationship between driver and device. Instead of allowing a driver direct hardware access to a device, USB limits communications between a driver and a device to four basic data

transfer types (i.e. bulk, control, interrupt and isochronous) implemented as a software interface provided by the host environment. This means a device must respond as expected by the system software layers or a driver will be unable to communicate with its device.

The *Universal Serial Bus Specification* identifies the requirements that all devices must meet to be compliant. In particular, **Chapter 9 USB Device Framework** outlines the standard requests and descriptors all USB devices must support. Class specifications add another layer of requirements directly related to how the software interacts with the capability performed by a device or interface which is a member of the class.

The illustration below indicates the relationships between a device and a host system. At the lowest level, the host controller physically communicates with USB hardware on the device through USB.

At the middle layer, USB system software uses the device abstraction defined in the *Universal Serial Bus Specification* to interact with the USB device interface on the device. This is the hardware or software which responds to standard requests and returns standard descriptors.

At the highest layer the driver uses an interface abstraction to interact with the function provided by the device. If the interface belongs to a particular class, the class specification defines this abstraction.
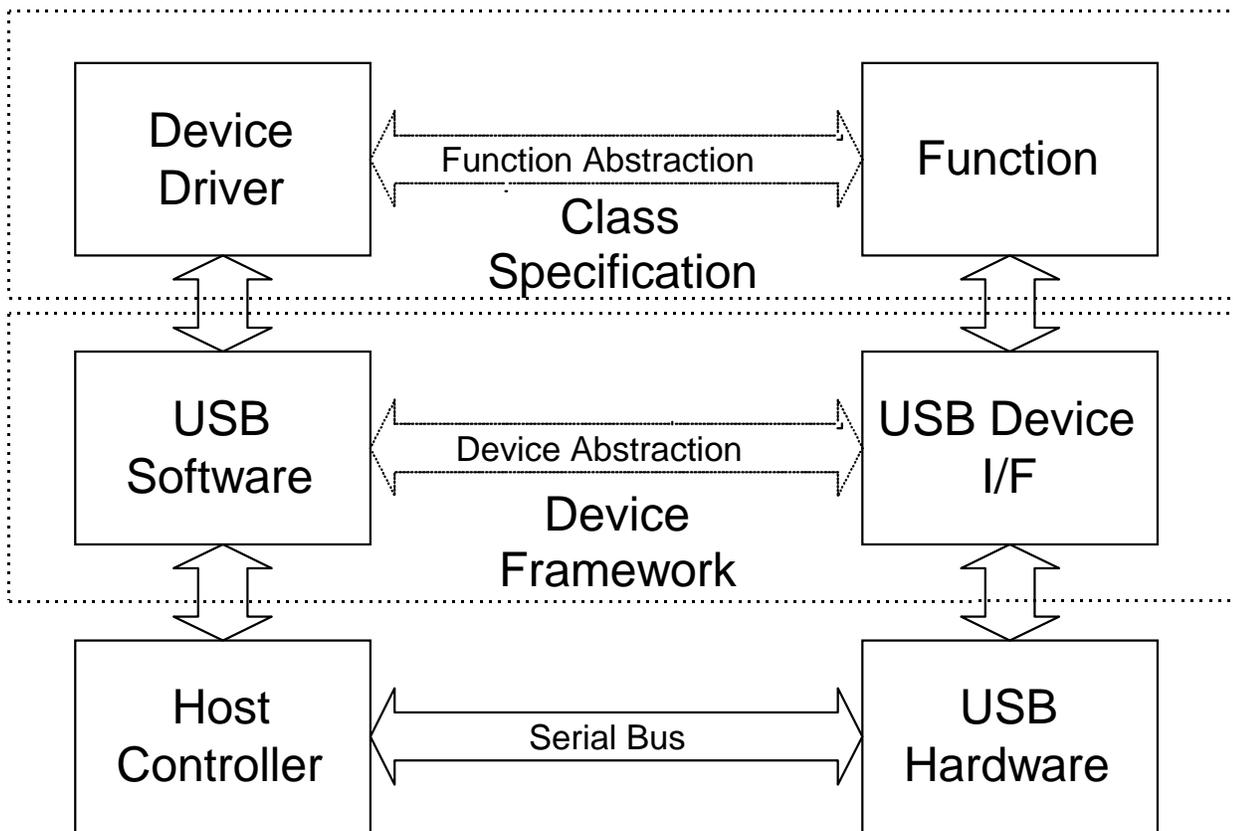
3.4 Connection versus Functionality

Classification of USB devices (and interfaces) does not necessarily follow the traditional approaches related to functionality. Instead, a USB class can define the manner in which a device communicates with the host. Because USB mandates that host software must communicate with a device or an interface via intervening software layers, how such a connection is accomplished must always be described.

For that reason USB classes must be based at least on how the device or interface connects to USB rather than just the attributes or services provided by the device. For example, the USB printer class does not identify how many paper trays or colors of ink a printer supports. Instead the printer class describes how a printer is attached to a host system, either as a single unidirectional output pipe or as two unidirectional pipes, one out and one in for returning detailed printer status.

USB classes also focus on the format of the data moved between host and device. While raw (or undefined) data streams may be used, the class may also identify data formats more specifically. Again using printers as an example, the output (and optional input) pipe may choose to encapsulate printer data as defined in another industry standard. The printer class provides a mechanism to return this information using a class specific command.

A USB class may also indicate enough information about the interface or device that a driver capable of managing the functionality, as well as the connectivity, of the interface is located and bound to that interface. For instance, the audio class defines

- the appearance of interfaces which are members of the audio class.

- how to identify the format of the data streams.

- class requests for setup and information retrieval.

and indicates that the capability provided by the interface is to consume or produce audio data. Other classes, like the communication class, identify how a broad range of capabilities are organized on such devices and largely rely on the functional and connectivity definitions provided by other classes to handle the specifics.

3.5 What is a Class Specification?

A Class Specification defines how devices using that class must behave. A device may use different classes for each interface it provides. It also may choose to implement interfaces which use no class definition at all. A complete class specification allows developers to develop adaptive software that is capable of operating any interface within the class. The class specification identifies features of devices that are implemented consistently across all devices using that class. In addition, the class specification anticipates variations in implementations and defines mechanisms to describe these variations for interrogation by adaptive drivers.

For example, a class specification may define a number of data formats which can be used by an endpoint. The specification then also details how the device reports which format(s) is used by a specific implementation. The class specification also describes if the format varies according to the selected device configuration or may vary dynamically based on host or external input. If selection of a particular format is possible, the class specification also describes how to administer the format.

3.6 Revision Numbering

Class specifications go through a development and review process prior to release, as described in the Device Working Group policies and procedures. Initially, a company or individual develops a conceptual proposal for a class specification. The proposal represents their view of the class. As the class specification is reviewed, and agreed to, by wider and wider audiences, the revision numbers increase. Intermediate versions of a class specification at a particular revision level are identified by an incrementing alpha suffix and use revision marks to indicate changes as the document moves forward.

3.7 Device Components

A USB device may be subdivided into a number of components, such as: device, configuration, interface and endpoint. Class specifications define how a device uses these components to deliver the functionality provided to the host system.

3.8 Specific Device Recognition

In some cases a host system uses device-specific information in a device or interface descriptor to associate a device with a driver. For example, the *idVendor* and *idProduct* fields in the device descriptor may uniquely identify a device and allow it to be associated with a driver. However, this style of driver association usually requires a driver written for a specific device. Class specifications attempt to provide device recognition to a host through identification of the device by class related affiliations.

3.9 Classes, Sub-Classes and Protocols

The standard device and interface descriptors contain fields that are related to classification: class, sub-class and protocol. These fields may be used by a host system to associate a device or interface to a driver, depending on how they are specified by the class specification.

Valid values for the class fields of the device and interface descriptors are defined by the USB Device Working Group. Valid values for the subclass and protocol fields are determined by each class working group and must be specified in the class specification.

3.10 Locating USB Drivers

Finding device drivers for USB devices presents some interesting situations. In some cases the whole USB device is handled by a single device driver. In other cases, each interface of the device has a separate device driver. The method for determining how device drivers are located and loaded needs to be defined generically for USB devices so that OS vendors and USB device providers are working within a common model. This section describes the common model for locating and loading USB device drivers.

Choosing a configuration determines the number of interfaces. The specific characteristics of the interface may be determined later via alternate settings. Typically, different device configurations are only required when a different power environment is to be used or a different number of interfaces required. They may in fact be viewed as a user configuration option, with all other configuration options being handled at the interface level.

Device drivers are searched for and located based on descriptor information from the USB device. The first search is based on information from the 'device' descriptor and looks for a driver that controls the whole device. The particular pieces of information (keys) used in the driver search are shown in the table below. Note that these are presented in priority order, and if no driver is found for a particular key, then the next key in order is used for the next search.

Key	Comments
idVendor & idProduct & bcdDevice	bcdDevice is the device's release number.
idVendor & idProduct	
idVendor & bDeviceSubClass & bDeviceProtocol	Only if bDeviceClass is FFH.
idVendor & bDeviceSubClass	Only if bDeviceClass is FFH.
bDeviceClass & bDeviceSubClass & bDeviceProtocol	Only if bDeviceClass is not FFH.
bDeviceClass & bDeviceSubClass	Only if bDeviceClass is not FFH.

If a driver is found in the above search, that driver is able to participate in choosing which configuration of the USB device should be used.

If no drivers are found from the above search, then system software is expected to choose an appropriate configuration for the USB device and then try to locate/load drivers for each interface in the chosen configuration. Keys for this driver search are based on information from both the 'device' and 'interface' descriptors. The table below shows the search keys in priority order.

Key	Comments
idVendor & idProduct & bcdDevice & bConfigurationValue & bInterfaceNumber	
idVendor & idProduct & bConfigurationValue & bInterfaceNumber	
idVendor & bInterfaceSubClass & bInterfaceProtocol	Only if bInterfaceClass is FFH.
idVendor & bInterfaceSubClass	Only if bInterfaceClass is FFH.
bInterfaceClass & bInterfaceSubClass & bInterfaceProtocol	Only if bInterfaceClass is not FFH.
bInterfaceClass & bInterfaceSubClass	Only if bInterfaceClass is not FFH.

3.11 Identifying Class and Vendor-Specific Requests and Descriptors

A USB Class Specification or a device vendor may define additional USB device requests. Class-specific requests are indicated by setting the *bmRequestType.Type* field of the setup packet to CLASS. The specific class defining the request is specified by the *bmRequestType.Recipient* field, that is the request class is the recipient class.

For instance, if the Recipient field is set to DEVICE, the *bDeviceClass* field of the Device Descriptor identifies the class defining the request. If the Recipient field is set to INTERFACE or ENDPOINT, the *bInterfaceClass* field of the Interface Descriptor identifies the class defining the request.

Vendor-specific requests are indicated by setting the *bmRequestType.Type* field of the setup packet to VENDOR. The specific vendor defining the request is specified by the *idVendor* field of the Device Descriptor.

The most significant bit of the *bDescriptorType* field is reserved for future use. For forward compatibility, this bit is handled as follows:

- Devices return this bit reset to zero when responding to a GET_DESCRIPTOR request.

- The host ignores the setting of this bit when it is returned by the GET_DESCRIPTOR request.

- The host resets this bit to zero before a SET_DESCRIPTOR request.

- Devices ignore this bit when receiving a SET_DESCRIPTOR request.

The next two most significant bits of the *bDescriptorType* field are used to indicate standard, class or vendor-specific descriptors. These bits use the same encodings as the *bmRequestType.Type* field of a USB device request setup packet. Because the upper three bits of the *bDescriptorType* field are used as described above, the maximum number of unique descriptors that may be defined for any category (standard, class or device-specific) is 32.

Whether class or vendor-specific USB device requests or descriptors are mandatory is determined by the defining class specification or vendor. When a device responds to a descriptor request with data that contains multiple descriptors, class or vendor-specific descriptors may be intermixed with standard descriptors. The position of the class or vendor-specific descriptor is used to associate that descriptor with prior descriptors. For example, if a class defines a descriptor extending the standard endpoint descriptor, a class specific endpoint extension descriptor would immediately follow each standard endpoint descriptor (endpoint descriptor, class-specific endpoint extension descriptor, endpoint descriptor, class-specific endpoint extension descriptor…).

The value used for the least significant five bits of a class or vendor-specific descriptor is defined by the appropriate class or vendor definition. That means a class or vendor-specific descriptor extending a standard descriptor is not required to use the same values as a standard descriptor they extend.

The standard GET_DESCRIPTOR request (with the *bRequestType.Type* field set to standard) is used to directly request class or vendor-specific descriptors. The class associated with such a request is determined by the class of the *bmRequestType.Recipient*. When the *bmRequestType.Recipient* field is set to INTERFACE or ENDPOINT, the *wIndex* field identifies the desired interface or endpoint. All endpoints within an interface use that interface's class, subclass and protocol.

3.12 Device Behavior: Alternate Settings

A USB host may issue the SET_INTERFACE command to select the alternate setting to be used with an interface. If the interface being set already has data queued for transmission to the host when the SET_INTERFACE command is issued, then this queued data shall be discarded.

4. Class Specification Format

This section describes the contents of the sections suggested for a class specification. Until a class specification reaches revision 1.0 it should contain the following disclaimer:

For Review and Discussion Only
Draft Document Subject to Revision or Rejection
Not For Publication or General Distribution

4.1 Title and Initial Pages

Each class specification begins with a title page that identifies the class specification by name. The title page includes the specification revision and release date.

The reverse side of the title page contains the following items:

- Scope of this Revision
- Revision History
- Intellectual Property Disclaimer
- Comment

For examples of the above, see the initial pages of this document.

4.2 Introduction

The introduction sets the overall goals for a class specification. It contains the following sub-sections:

- Purpose
- Scope
- Related Documents
- Terms and Abbreviations.

The Purpose sub-section describes why the class specification is being created.

The Scope sub-section describes devices that are included within the class specification and may specifically identify devices which either are not intended to be a part of the class or are not currently targeted for support.

The sub-section for Related Documents identifies other document sources that contribute to the definition of the class. If the class re-uses other industry standards, specific citations are required.

4.3 Management Overview

This section is a one or two page overview that allows readers to understand the class and the range of implementation possibilities without requiring an exhaustive review of the entire document.

4.4 Functional Characteristics

This section of the class specification provides a description of each of the functional characteristics provided by devices belonging to the defined class.

4.5 Operational Model

This section describes how the device is expected to interact with a host system. For example, this section might explain how and why a host system sends commands through the default pipe to select class specific actions on an interrupt pipe.

4.6 Descriptors

The *Universal Serial Bus Specification* defines a number of standard descriptors. This section defines how the class uses those standard descriptors (e.g. values for the class, subclass and protocol fields of the device and interface descriptors) and any additional descriptors defined by the class (class-specific descriptors).

4.7 Requests

The *Universal Serial Bus Specification* also defines a number of standard requests that all devices must support. This section defines how the class uses those standard requests, if they differ from the standard implementations. If a class specification adds additional class-specific requests, they are also described in this section.

4.8 Device Components

This defines how configurations, interfaces and endpoints may be defined to implement this class.

4.9 Electrical, Protocol and Transport Considerations

A device class may choose to restrict or expand the use of features defined in the *Universal Serial Bus Specification* in standard areas such as power or protocol. Such variations are described here.

4.10 Class Interactions

A class may choose to make extensive use of other classes' definitions to implement its capabilities. The requirements of such interactions are described here.

4.11 Appendices

If required, a class specification may provide appendices to list tabular information or supplement the basic specification. For example, if a class specification added a number of class-specific requests or descriptors, an appendix might be used to provide tables illustrating the numeric constants used for specific requests or descriptors.

5. Common Attributes and Services Overview

This section presents an overview of attributes and services, which are not covered in the *Universal Serial Bus Specification,* but which can be used by more than one class.

5.1 Synchronization

The *Universal Serial Bus Specification* identifies several types of synchronization between sources and sinks of digital data streams. What is not defined in that specification is a method of reporting the synchronization requirements of a specific endpoint. The specification also does not describe how synchronization feedback information is returned by a device; for example, which endpoint reports feedback information and what is the format used for reported feedback information. Refer to *USB Feature Specification: Synchronization* for details.

5.2 Dynamic Interfaces

The *Universal Serial Bus Specification* describes devices being configured as a part of the initialization process and interfaces which have the same, known, capabilities available subsequently that they had at configuration time. Some devices require a change in interface definition due to an event external to the host or device.

For example, a telephone call might be received by a multimedia modem. Due to the nature of the call, the device might be able to determine that the data being received was audio, or fax or unformatted data. To utilize the appropriate class definition, the interface providing the data from the modem might require a change from its original setting. Dynamic interfaces allow the device to report the need for this change and describe how a host system may determine the new interface type and request any necessary changes. Refer to *USB Feature Specification: Dynamic Interfaces* for details.

5.3 Associations

Extending the example in the previous section, when a call is received by a multimedia modem it may actually contain multiple data elements. The call might have voice and video information. Following USB conventions, this would require two interfaces, one defined by an imaging class and another by an audio class. However, now the interfaces would actually be related (or associated) because both were tied to the same call.

Associations provide the necessary definition to describe how a device reports the interrelation of multiple interfaces and allows a host to react accordingly. Refer to *USB Feature Specification: Associations* for details.

5.4 Shared Endpoints

The *Universal Serial Bus Specification* allows only the default endpoint to be shared between interfaces on a device. This was a simplifying assumption to reduce coupling between interfaces. However, as additional classes were defined, it became clear that several interfaces on a device could have very similar requirements on an endpoint, which would allow it to be shared among those interfaces. Such sharing would reduce the overall number of endpoints required, and thus the overall cost of the device.

For instance, the specification of synchronization mechanisms and reuse of class specifications to allow easy driver binding could substantially increase the number of endpoints required for some devices. Devices which require feedback would need a substantial number of endpoints since each endpoint requiring synchronization may require a separate endpoint to report feedback information.

In addition, some devices may actually be collections of interfaces that use very simple and low bandwidth data reporting mechanisms, such as an interrupt endpoint, for a common activity such as event notification. If this interrupt endpoint could be shared by multiple interfaces, the device could use fewer endpoints and fewer endpoints would need to be scheduled. Refer to *USB Feature Specification: Shared Endpoints* for details.

5.5 Interface Power Management

A method for providing power management to an interface on a USB device is not described in the *Universal Serial Bus Specification*. Interface power management enables the host software to manage power savings and remote wake-up behavior independently on each separate interface of a device. An example use of interface power management is its application to composite devices. A *composite device* is a USB device that has more than one interface and each interface is controlled by a different device driver running on the host (or by a different instance of the same driver running on the host). For example,

- An audio-visual device can have two interfaces, audio (an Audio class interface) and video (an Imaging class interface). Independent power management of each interface enables the host to put the video interface in a power saving mode when only the audio interface is being used.

- A telephony device can have three interfaces: audio, a keypad (a HID class interface), and a modem (a Communications class interface). The host can use interface power management to put the modem in a low-power wake-enabled mode when only the keypad and audio interfaces are being used.

An interface power descriptor can provide the following benefits:

- USB devices can implement a range of low-power modes, not just Suspend.

- Each interface of a device with multiple interfaces can be power-managed independently.

- Each interface of a device with multiple interfaces can be wake-enabled independently.

- Self-powered devices can conserve energy.

Refer to *USB Feature Specification: Interface Power Management* for details.

5.6 Default Notification Pipe

Many class specifications have built class-specific *Device-to-Host* asynchronous notification controls based on interrupt pipes. These are always point-to-point solutions and define the format of the data and a set of class-specific notification messages. The Default Notification Pipe provides a common, standard solution for moving device events to the appropriate level of system software. Refer to *USB Feature Specification: Default Notification Pipe*.