Open Universal Serial Bus Driver Interface (OpenUSBDI) Specification

Revision 1.0

July 17, 2000

Revision History

Rev	Date	Filename	Comments
1.0	5-May-00	usbd10.doc	Update version to 1.0, remove "Review and Discussion Only" remarks from document.
1.0rc1	25-April-00	USBDI_1rc1.doc	More UDI –related changes
0.9b	22-October-99	USBDI_09b.doc	Made yet more UDI 1.0 spec changes.
0.9a	17-August-99	USBDI_09a.doc	Brought spec up to date with the UDI 1.0 spec. Added usbdi_device_speed ops.
0.9	21-June-99	USBDI_09.doc	Promoted the spec to .90
0.8	08-May-99	USBDI_08.doc	Promoted the spec to .80
0.8rc	03-May-99	USBDI_08rc.doc	Made updates from 5/6 teleconf
0.7d	27-April-99	USBDI_07d.doc	Made updates from 4/14 F2F
0.7c	9-April-99	USBDI_07c.doc	Made updates from 3/29 F2F, Tech writers comments
0.7b	25-Mar-99	USBDI_07b.doc	Added sections on async notification, device state, UDI overview, and lots of pictures. Made lots of changes based of two months of con-calls and reviews.
0.7a	15-Jan-99	USBDI_07a.doc	Reformatted document.
0.72	6-Jan-99	USBDI_072.DOC	Incorporated usbdi.h functions, structs, and typedefs
0.71	23-Oct-98	USBDI_071.DOC	
0.7	04-Sep-98	USBDI_07.DOC	Move to 0.7 with formation of Working Group and incorporate comments from August 26 & 27 face to face meetings. Pipes are opened in interface bundles, not individually. Interfaces may be opened by only one driver at a time. Functions are organized (and named) based on the need for UDI support in the OS. Changed many USBDI functions to use the non-blocking callback on completion method of operation. Changed the name of the spec to OpenUSBDI.
0.6a	12-Aug-98	USBDI_06a.DOC	Add UDI functions and place holders for USB Class appendixes
0.6	10-Aug-98	USBDI_06.DOC	Initial draft in new format incorporating “USBDI Functional Specification (Rev. .7) 3/24/98

Please send comments via electronic mail to either Janet.Schank@compaq.com ***or*** Aaron.Biver@compaq.com**.**

Contributors

Wendy Adams	Hewlett Packard
Aaron Biver	Compaq Computer Corporation
Mark Evenson	Hewlett Packard
Kurt Gollhardt	SCO
Anish Gupta	Sun Microsystems
John Howard	Intel
Forrest Kenney	Compaq Computer Corporation
James Partridge	IBM
Janet L. Schank	Compaq Computer Corporation
Kevin Quick	Interphase Product Development

Table of Contents

1 Introduction

1.1 Purpose

This document specifies the design parameters for a device driver using the Open Universal Serial Bus Driver Interface (OpenUSBDI) Specification.

1.2 Scope

This document is intended to provide enough information to allow a software developer to create device drivers capable of supporting compliant USB devices on operating systems implementing support for the Open USB Driver Interface.

1.3 Related Documents

Compaq, Intel, Microsoft, NEC, *Universal Serial Bus Specification*, Version 1.1, September 23, 1998

Project UDI, *UDI Core Specification*, Version 1.0, September 2, 1999

SystemSoft Corporation, Intel Corporation, *Universal Serial Bus Common Class Specification*, Version 1.0, December 16, 1997

Compaq, Microsoft, National Semiconductor, *OHCI Open Host Controller Interface Specification for USB*, Release 1.0a, January 20, 1997

Intel, *Universal Host Controller Interface (UHCI) Design Guide*, Revision 1.1, March 1996

1.4 Terms and Abbreviations

Term	Description
ABI	Architected Binary Interface. This is a set of binary bindings for a programming interface specification such as the OpenUSBDI Specification. (When applied to applications rather than system programming interfaces, ABI is usually interpreted as Application Binary Interface.)
API	Architected Programming Interface. This is a programming interface defined in a OpenUSBDI specification; e.g., a function call interface or structure definition with associated status or function codes, as well as associated semantics and rules on the use of the interfaces. (When applied to applications rather than system programming interfaces, API is usually interpreted as Application Programming Interface.)

Term	Description
Bulk Transfer	Non-periodic, large bursty communication typically used for data that can use any available bandwidth and that can be delayed until bandwidth is available.
Channel	A bidirectional communication channel between two drivers, or between a driver and the environment. Channels allow code running in one region to invoke channel operations in another region. Example of channels include the bind channel between a logical-device driver instance and the USBD instance, and the management channel between a driver instance and the Management Agent.
Class Driver	An adaptive driver based on a class definition.
Control Transfer	Non-periodic, bursty, host-software-initiated request/response communication typically used for command/status operations.
Control Block (CB)	A data structure that gives details about OpenUSBDI requests. There are different control blocks types for different request types. Typically control blocks contain flag fields, references to data buffers, etc.
Device	A logical or physical entity that performs one or more functions. The actual entity described depends on the context of the reference. At the lowest level, a device may be a single hardware component, such as a memory device. At a higher level, a device may be a collection of hardware components that perform a particular function, such as a USB interface device. At an even higher level, the term “device” may refer to the function performed by an entity attached to the USB, such as a data/FAX modem device. Devices may be physical, electrical, addressable, or logical. When used as a non-specific reference, a USB device is either a hub or a function.
GIO	Generic I/O Metalanguage. See the UDI Core Specification for more information.
HC	A USB Host Controller.
HCD	A USB Host Controller Driver.
Hub	A USB device that provides attachment points to the USB.
Interrupt Transfer	Non-periodic, low frequency, and bounded-latency small data transfers that are typically used to handle service needs.
Isochronous Transfer	Periodic, continuous communication between host and device typically used for time relevant information. This transfer type also preserves the concept of time encapsulated in the data.
Logical-Device	One or more USB interfaces that function together as a single entity.

Term	Description
Logical-Device Driver (LDD)	A driver that resides above the Universal Serial Bus Driver that controls devices with certain functional characteristics in common. This may be a single interface of a USB device or it may be a group of interfaces. In the case of a group of interfaces, the "Common Class Logical-Device Feature Specification" shall be implemented by the device.
LDD Instance	A set of one or more regions, all belonging to the same LDD, that are associated with a particular instance of the driver's device. There may be multiple instances of a given LDD, one for each physical device controlled.
Management Agent (MA)	The MA is an abstract entity within the UDI environment; it represents the environment's control and configuration mechanisms.
Metalanguage	The communication protocol used by two or more cooperating modules. A metalanguage includes definitions for associated channel operations, control block structures, and service calls, as well as bindings to the use of UDI trace events and the definition of various types of UDI instance attributes. E.g., the OpenUSBDI Metalanguage is used for communication between USB logical-device drivers and the USB driver layer. When referring to a metalanguage used by a particular type of driver the adjectives "top-side" and "bottom-side" are sometimes applied: e.g., the OpenUSBDI Metalanguage is the bottom-side metalanguage for USB logical-device drivers.
OHCI	Open Host Controller Interface
OpenHCI	Open Host Controller Interface
Pipe Channel	Channel between the LDD and a USB device's endpoint.
UDI	See Uniform Driver Interface.
Uniform Driver Interface (UDI)	Allows device drivers to be portable across both hardware platforms and operating systems without any changes to the driver source. With the participation of multiple operating systems, as well as platform and device hardware vendors, UDI is the first interface that is likely to achieve such portability on a wide scale. UDI provides an encapsulating environment for drivers with well-defined interfaces that isolate drivers from OS policies and from platform and I/O bus dependencies. This allows driver development to be totally independent of OS development. In addition, the UDI architecture insulates drivers from platform specifics such as byte ordering, DMA implications, multi-processing, interrupt implementations and I/O bus topologies.
UHCI	Universal Host Controller Interface
USB device	See "device".

Term	Description
USBD	Universal Serial Bus Driver; operating system specific driver that provides an interface between OpenUSBDI and the host controller(s).
OpenUSBDI	Open Universal Serial Bus Driver Interface or Open USB Driver Interface. The interface that this document describes.

2 Management Overview

This specification defines all functions needed by USB logical-device drivers (LDDs) to communicate with the operating system's USB driver stack. The Open USB Driver Interface (OpenUSBDI) is designed to work in conjunction with the Uniform Driver Interface (UDI) Core Specification. This approach allows compliant drivers, developed for USB peripherals, to be easily transported from one operating system to another. An operating system that uses these portable drivers shall provide the following components: a UDI environment that is compliant with the UDI Core Specification; an OpenUSBDI environment that provides the support spelled out in this specification, and a USB driver stack (USBD).

Device drivers written using UDI offer source-level portability across operating systems, and binary level portability across platforms that support the same ABI.

This specification does not define UDI but does depend on it. This specification only defines the interface that is specific to USB device drivers.

For more information on UDI see the Project UDI web page, http://www.project-UDI.org.

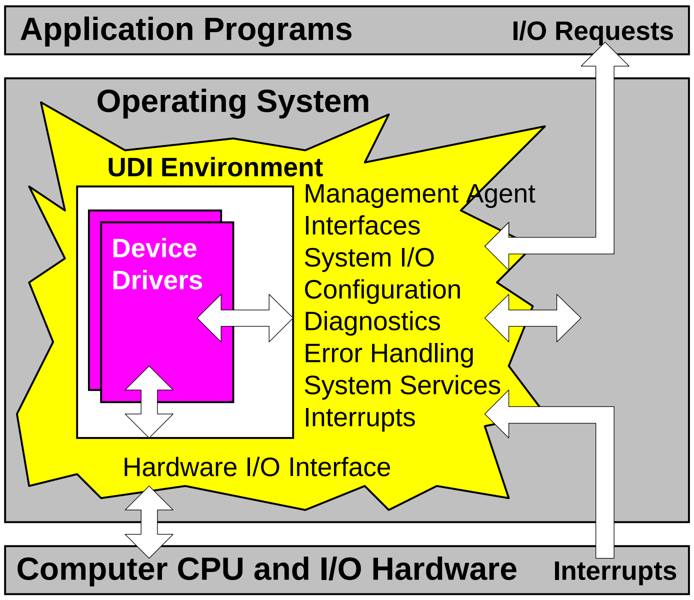

Figure 1: UDI Driver Environment

Figure 1 shows where within an operating system the UDI environment resides.

The OpenUSBDI specification defines the communication interface between a USB Logical-Device Driver (LDD) and the operating system's USB driver (USBD) stack.

All OS services not related to USB required by an LDD shall be performed using operations based on the UDI Core Specification.

USB LDDs shall use the appropriate UDI metalanguage when exchanging device information with the OS and when communicating with applications. Where no specific metalanguage is defined the "Generic I/O Metalanguage"[1] may be used.

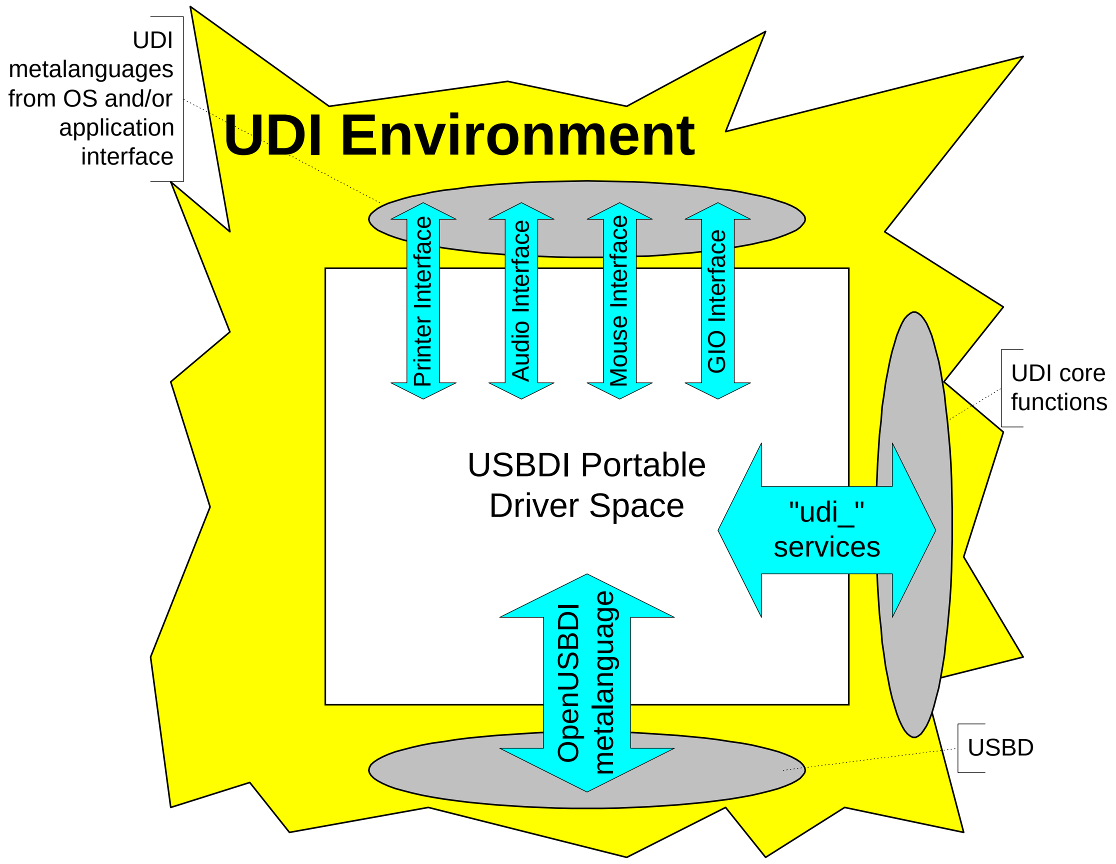

Figure 2: OpenUSBDI Portable Environment

[1] Defined in the UDI Core Specification

Figure 2 depicts the environment of a portable LDD, and the services available to it. In order to remain completely portable, the LDD shall limit itself to the following:

- **UDI Metalanguages and/or Application Interface:** A "pass-through" interface is defined within the UDI Core Specification, "Generic I/O Interface", that allows vendor-unique drivers the ability to communicate with vendor-unique applications. For each USB Device Class[2], a specific operating system interface is defined. These interfaces are published as separate documents from the UDI Core Specification. These interfaces are not specific to USB devices, but are general for the type of device. For example, both a USB printer class driver and a non-USB printer driver would use the UDI printer interface.

- **udi_ services:** Non-USB specific, operating system services (such as buffer allocation and timers) are defined by the UDI Core Specification. A USB LDD shall use only UDI core functions for these tasks to be 100% source code portable. All core UDI functions are prefixed with "`udi_`".

- **OpenUSBDI Metalanguage:** The USBD is the driver layer separating the LDD from the host controller driver. All communications between LDDs and host controllers pass through the USBD layer. The OpenUSBDI metalanguage is the visible functional interface for LDDs communicating with the USBD. This specification defines all functions needed by LDDs to communicate with the operating system's USBD. All functions that are part of the OpenUSBDI metalanguage are prefixed with "`usbdi_`".

- **USBD**: Operating system specific USB driver that communicates on its top side with LDDs via the OpenUSBDI metalanguage and communicates with USB host controllers on its bottom side.

A LDD that adheres to this specification and uses only core UDI functions will be 100% source code compatible on all operating systems that implement OpenUSBDI and the UDI Core Specification.

Some environments may implement the OpenUSBDI interfaces defined in this specification without implementing a complete UDI environment. Portable USB LDDs will have to be modified to use environment specific service calls in order to run in such environments.

[2] See http://www.usb.org for USB device class specifications.

3 Functional Overview

Figure 3 shows an example USB system. The system is composed of several layers, ranging from the hardware layer to USB logical-device drivers.

Following is a description of each critical layer in a USB system. Refer to figure 3 as needed.

Layer	Description
Logical-Device Drivers	This layer performs the task of controlling specific USB devices (vendor-unique) and specific USB device classes (printer, mass storage, etc.).
USB Driver	This layer provides an organized method of transferring data between the Host Controller Driver (HCD) and the USB Logical-Device drivers. The interface between the LDD and USBD is the set of procedures and data types defined by this specification. The USBD implementation is not addressed by this specification.
Host Controller Driver	This layer has an intimate knowledge of the host controller hardware. It provides a means for higher layers to convey information to devices, and vice versa. Because of the differences among host controllers, the HCDs are not interchangeable.
Hardware	This layer consists of physical wires, the host, and the USB devices. The host requires a host controller and each device requires a device controller. There are currently two common implementations of the host controller: Open Host Controller Interface (OHCI) and Universal Host Controller Interface (UHCI).

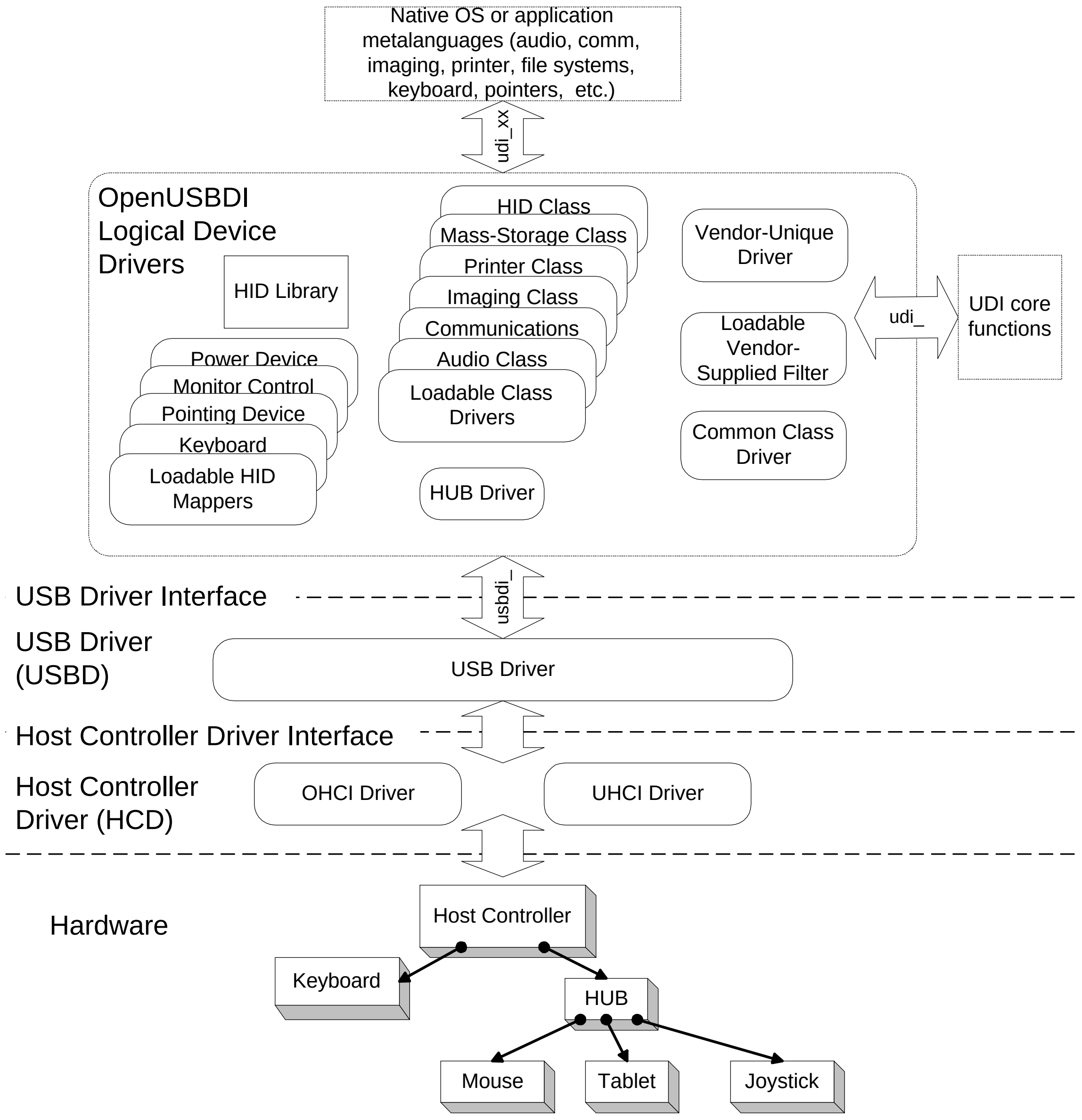

Figure 3: USB Environment

4 Functional Characteristics

4.1 Versioning

All functions and structures defined in the OpenUSBDI Driver specification are part of the "open_usbdi" interface, currently at version "0x100". A driver that conforms to and uses the OpenUSBDI Driver Specification, Version 0x100, must include the following declaration in its `udiprops.txt` file (see Chapter 31, "Static Driver Properties", of the UDI Core Specification):

```
requires open_usbdi     0x100

requires udi            0x100
```

Before including any UDI header files, the LDD shall define the preprocessor symbol, OPEN_USBDI_VERSION, to indicate the version of the OpenUSBDI Driver specification to which it conforms. For this version of the specification, OPEN_USBDI_VERSION must be set to 0x100:

```
#define OPEN_USBDI_VERSION    0x100
#define UDI_VERSION           0x100
```

A portable implementation of the OpenUSBDI Metalanguage shall include a corresponding "provides" declaration in its `udiprops.txt` file and must also define OPEN_USBDI_VERSION.

As defined in Section 31.4.6, "Requires Declaration," on page 31-6 of the UDI Core Specification, the two least-significant hexadecimal digits of the interface version represent the minor number; the rest of the hex digits represent the major number. Versions that have the same "major version number" as an earlier version shall be backward compatible with that earlier version (i.e. a strict superset).

4.2 Header Files

Each device driver source file shall include the file **open_usbdi.h** after it includes **udi.h**, as follows:

```
#include <udi.h>
#include <open_usbdi.h>
```

The **open_usbdi.h** file includes defines and function declarations for the OpenUSBDI metalanguage and for the Universal Serial Bus Specification

4.3 Bindings to the UDI Core Specification

4.3.1 Static Driver Properties Bindings

Some of the bindings from the static driver properties are defined in Section 4.1, "Versioning". This includes the definition of the relevant interface name(s) (i.e. the <interface_name> parameter on the "requires" and "provides" and other property declarations), and the definition of the interface version number for this version of this Specification.

The driver category to be used with the "category" declaration (see Section 31.5.3, "Category Declaration," on page 31-9 of the UDI Core Specification) by a portable implementation of the OpenUSBDI Metalanguage shall be "USB Logical Drivers".

4.3.2 Instance Attributes Bindings

In each of the attribute tables below, the **ATTRIBUTE NAME** is a null-terminated string (see "Instance Attribute Names" on page 16-1 of the UDI Core Specification); the **TYPE** column specifies an attribute data type as defined in "udi_instance_attr_type_t" on page 16-7 of the UDI Core Specification; and the **SIZE** column specifies the valid sizes, in bytes, for each attribute.

4.3.2.1 Enumeration Attributes

The driver that enumerates the USB logical device must create the following enumeration attributes and pass them to the Management Agent in the `attr_list` parameter of the `udi_enumerate_ack` operation (see "Enumeration Operations" on page 25-12 of the UDI Core Specification).

Enumeration attributes that refer to strings returned by USB logical devices shall include only the UTF-8 encoded data portion of the returned string descriptor.

ATTRIBUTE NAME	TYPE	SIZE (in bytes)	DESCRIPTION
usb_vendor_id	UDI_ATTR_UBIT32	4	Vendor ID (assigned by the USB Device Working Group)
usb_product_id	UDI_ATTR_UBIT32	4	Product ID (assigned by manufacturer)
usb_release_num	UDI_ATTR_UBIT32	4	Device release number
usb_device_class	UDI_ATTR_UBIT32	4	Device class code (assigned by the USB Device Working Group)
usb_device_subclass	UDI_ATTR_UBIT32	4	Device subclass code (assigned by the USB Device Working Group) These codes are qualified by the value of Device_Class in the device descriptor.
usb_device_protocol	UDI_ATTR_UBIT32	4	Protocol code.
usb_configuration_value	UDI_ATTR_UBIT32	4	Value to use as an argument to "Set Configuration" to select this configuration.

usb_serial_number	UDI_ATTR_STRING	UDI_MAX_ ATTR_SIZE (64)	The first UDI_MAX_ATTR_SIZE ASCII bytes of the serial number of the USB device[3].
usb_product_string	UDI_ATTR_STRING	UDI_MAX_ ATTR_SIZE (64)	The first UDI_MAX_ATTR_SIZE ASCII bytes of the product string of the USB device.[3]
usb_manufacturer_string	UDI_ATTR_STRING	UDI_MAX_ ATTR_SIZE (64)	The first UDI_MAX_ATTR_SIZE ASCII bytes of the manufacturer string of the USB device.[3]
usb_identifier_attr	UDI_ATTR_UBIT32	4	This value is the Identifier attribute as specified in section 4.3.2.2.1 on page 19
usb_interface_number	UDI_ATTR_UBIT32	4	Zero-based index identifying the selected interface in the array of concurrent interfaces supported by this configuration.
usb_interface_subclass	UDI_ATTR_UBIT32	4	Interface subclass
usb_interface_protocol	UDI_ATTR_UBIT32	4	Interface protocol code
usb_interface_class	UDI_ATTR_UBIT32	4	Interface class

4.3.2.2 Generic Enumeration Attributes

There are four generically accessible enumeration attributes: "`identifier`", "`address_locator`", "`physical_locator`", and "`physical_label`". These attributes, of type UDI_ATTR_STRING, are defined so as to allow environments to use these attributes in generic algorithms to identify and compare information about the devices in the system. This is useful in keeping the UDI environment isolated from the specifics of metalanguages and bus bindings.

4.3.2.2.1 identifier attribute

The "`identifier`" attribute is a computed value. The attribute is computed by taking the three optional strings a device may supply (product string, manufacturer string, and serial number string), converting the strings to UTF-8 encoding, and building a new string up to 768 bytes in length. This string is run through a FCS computation that results in a 32 bit identifier. This FCS value is not guaranteed to be unique. To uniquely identify a device the identifier attribute needs

[3] While USB devices return Unicode strings, UDI attribute management facilities require attribute strings to be UTF-8 encoded. Therefore, the parent LDD shall translate the product, manufacturer and serial number strings to a UTF-8 encoding scheme before reporting the child's attributes.

to be combined with the address locator and the physical locator attributes. The FCS algorithm is discussed in Appendix D.

If the USB device does not supply any of the strings (product string, manufacturer string, or serial number) the identifier shall consist of the vendor id in bits 16 through 31 and the product id in bits 0-15.

4.3.2.2.2 address_locator attribute

The "`address_locator`" attribute for a USB device is a unique value that identifies the USB device's interface. This unique value is formed by combining the device's USB address ("`A`"), and the interface number ("`N`") as follows:

```
address_locator = AANN
```

where `AA` is a two-digit upper-case hexadecimal-encoded ASCII representation of the USB address and `NN` is a two-digit upper-case hexadecimal-encoded ASCII representation of the interface number.

4.3.2.2.3 physical_locator attribute

For USB devices, the "`physical_locator`" attribute encodes the location of the device on the USB device tree based on a sequence of hub port numbers starting with the port number from the root hub (tier 0). Port numbers begin with one and can reach 255. The maximum number of hubs between the host controller and a USB device is five, therefore the length of the "`physical_locator`" for USB is six bytes (one for each hub plus one for the host controller).

The value of the "`physical_locator`" attribute is formed by left-shifting the port number of each hub a number of bytes equal to that hub's tier number and OR'ing the result. The "`physical_locator`" for the device shown below would be:

```
physical_locator = (1 << 0)  || /* First port on root hub (Tier 0)*/
                   (3 << 8)  || /* Third port on hub in Tier 1    */
                   (3 << 16) || /* Third port on hub in Tier 2    */
                   (7 << 24);   /* Seventh port on hub in Tier 3  */
```

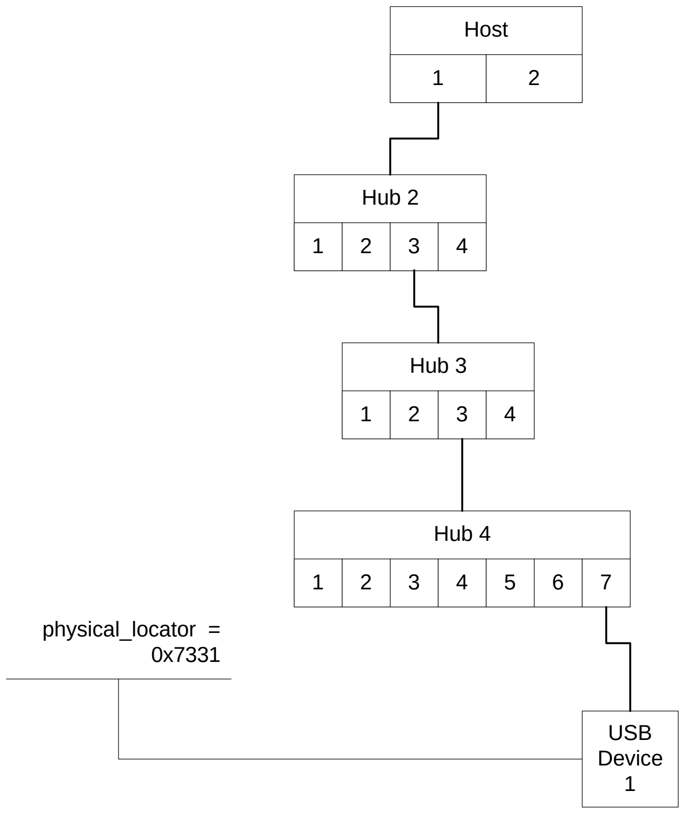

Figure 4: USBDI physical_locator example

4.3.2.2.4 physical_label attribute

No “`physical_label`” attribute is defined for the OpenUSBDI metalanguage.

4.3.2.3 Filter Attributes

The OpenUSBDI Metalanguage does not provide filter attributes.

4.3.2.4 Parent-Visible Attributes

There are no parent-visible attributes defined by the OpenUSBDI Metalanguage.

4.3.3 Trace Event Bindings

The following defines the rules and conventions in the USB Metalanguage for the use of the metalanguage selectable trace events (see the "USB-Selectable Trace Events" #defines in `udi_trevent_t` on page 18-3 of the UDI Core Specification).

- `UDI_TREVENT_META_SPECIFIC_1`
  This trace event should be used to track the state of requests. Events pertaining to states of requests or actions performed on them should be logged with this trace event. This includes transfers, completions, and abortions.
  Message text should include: a pointer to the control block, the number of bytes that were (or will be) transferred, the status (if the event is a completion), and whether this is a transfer, completion, or abortion.

- `UDI_TREVENT_META_SPECIFIC_2`
  This trace event should be used to track the state of pipes. Events pertaining to states of pipes or actions performed on them should be logged with this trace event.

- `UDI_TREVENT_META_SPECIFIC_3`
  This trace event should be used to track the state of interfaces. Events pertaining to states of interfaces or actions performed on them should be logged with this trace event.

- `UDI_TREVENT_META_SPECIFIC_4`
  This trace event should be used to track the state of endpoints. Events pertaining to states of endpoints or actions performed on them should be logged with this trace event.

- `UDI_TREVENT_META_SPECIFIC_5`
  Information that pertains to the USB device as a whole should be logged with this trace event. This may include device resets, device configuration changes, device state information, and asynchronous events received with a LDD's *usbdi_async_event_ind_op_t* function.

4.4 Metalanguage State Diagram

See "Driver Instantiation" on page 25-2 of the UDI Core Specification for the general configuration sequence of UDI drivers. See "Management Metalanguage States" on page 25-36 of the UDI Core Specification for details on the Management Metalanguage states.

The following state diagram shows the OpenUSBDI metalanguage state diagram, which illustrates the set of states specific to use of the OpenUSBDI metalanguage.

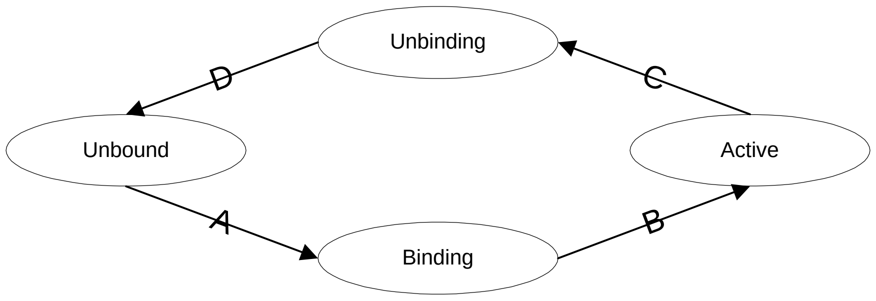

Figure 5 - USBDI Metalanguage State Diagram

Event	Operation
A	usbdi_bind_req
B	usbdi_bind_ack_op_t
C	usbdi_unbind_req
D	usbdi_unbind_ack_op_t

Table 1 - USBDI Metalanguage State Transition Table

State	Decription	Next State
Unbound	A channel in the unbound state has been established between the LDD and the logical device but has not yet been initialized in those regions for general use. The LDD side of the channel should initiate the `usbdi_bind_req` operation when in this state.	Binding
Binding	This state occurs when the LDD side of the channel has initiated a bind operation and is waiting for the USBD side of the channel to complete its initialization and acknowledge that bind request.	Active
Active	The LDD has bound to the logical device, and may interact with it. This interaction may include opening and closing interfaces and pipes, reading descriptors, and/or resetting the device.	Unbinding
Unbinding	The LDD is in the process of unbinding from the logical USB device via the `usbdi_unbind_req` operation.	Unbound

Table 2 – USBDI Metalanguage State Descriptions

4.5 Channels

UDI Channels are used to communicate between OpenUSBDI Logical-Device drivers (LDD) and the device endpoints via the operating system's USB driver (USBD).

An LDD shall maintain a channel for each USB interface for which it is responsible[4]. The LDD shall also maintain a channel for each endpoint controlled by each interface; these *pipe channels* correspond to pipes between the LDD and each endpoint.

4.6 Completion Status Values

All but two of the completion status values for OpenUSBDI control blocks (CBs) are described in the UDI Core Specification. The two OpenUSBDI unique status values are:

[4] Any device that uses multiple USB interfaces shall comply with the Common Class Logical-Device Feature Specification.

```
#define USBDI_STAT_NOT_ENOUGH_BANDWIDTH   (UDI_STAT_META_SPECIFIC|1)
#define USBDI_STAT_STALL                  (UDI_STAT_META_SPECIFIC|2)
```

4.7 Device Driver Flow

In order to transfer data to and from its associated device, an LDD shall first initialize itself, bind to the LDD's parent driver, select an appropriate interface that may involve reading descriptors from the logical-device, and open the selected interface. At this point the logical-device is ready for use. When the logical-device is no longer in use, the LDD shall go through the reverse process of closing interfaces and unbinding from the LDD's parent driver. These steps are described in detail and illustrated in the paragraphs and figures that follow.

In these figures, the following conventions are used. The three columns labeled "UDI", "LDD", and "USBD" represent the UDI environment, the OpenUSBDI Logical-Device Driver, and the USBD respectively. The arrows show the control flow from module to module, with the stop sign showing where the flow ends. For example, Figure 6 shall be read as follows:

1. The UDI Management Agent calls the LDD supplied function *ldd_bind_to_parent_req()*, so the control flows from UDI to LDD.

2. Inside the *ldd_bind_to_parent_req()* routine, the LDD calls *usbdi_bind_req()*, which invokes the corresponding function in the USBD, so control flows from the LDD to the USBD. The *usbdi_bind_req()* routine returns immediately to the LDD after initiating the request, which will be completed asynchronously at a later time. This is illustrated by the arrow going back from *usbdi_bind_req()* to *ldd_bind_to_parent_req()*. This control flow then terminates in the LDD, as illustrated by the stop sign.

3. At some point later, the results of the bind are returned to the LDD via the *ldd_bind_ack()* channel operation, invoked by the USBD. This results in a call to the LDD's *ldd_bind_ack_op_t* entry point routine.

For specific LDD examples refer to the sample driver provided in "Appendix B: Sample Driver (usbdi_printer.c)" of this specification.

4.7.1 Logical-Device Driver Initialization

All LDDs shall provide an *udi_init_info* struct. This structure is read by the Management Agent when the LDD is loaded. This structure is set up by the LDD with the following information (see “Initialization”in Chapter 10 of the UDI Core Specification for more details):.

1. the LDD’s scratch[5] space requirement for each type of OpenUSBDI control block (CB),

2. the LDD interface-related operation functions and the pipe-related operation functions, and

3. the LDD’s primary and secondary regions.

[5] Refer to the UDI Core Specification for the definition of this term.

4.7.2 Interface Binding

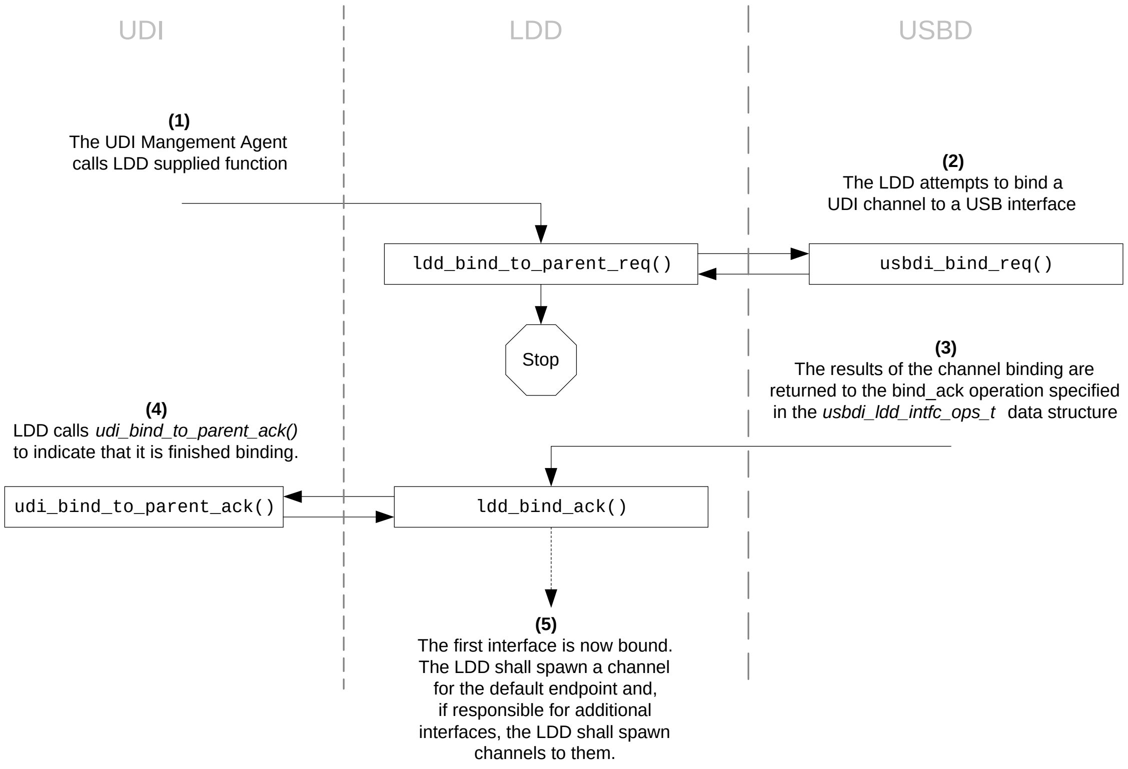
UDI
LDD
USBD
(1)
The UDI Mangement Agent calls LDD supplied function
(2)
The LDD attempts to bind a UDI channel to a USB interface
ldd_bind_to_parent_req()
usbdi_bind_req()
Stop
(3)
The results of the channel binding are returned to the bind_ack operation specified in the usbdi_ldd_intfc_ops_t data structure
(4)
LDD calls udi_bind_to_parent_ack() to indicate that it is finished binding.
udi_bind_to_parent_ack()
ldd_bind_ack()
(5)
The first interface is now bound. The LDD shall spawn a channel for the default endpoint and, if responsible for additional interfaces, the LDD shall spawn channels to them.

Figure 6: Interface Binding

After the LDD is initialized, the UDI Management Agent may bind one or more instances of the LDD to the USBD. For each instance, the LDD’s *udi_bind_to_parent_req_op_t* function shall be called. This function is responsible for binding USB interface channels for all instances managed by this LDD instance, by using *usbdi_bind_req()*.

The LDD completes the bind sequence when it receives a *ldd_bind_ack()* call. At this point, the LDD shall spawn its end of any additional interface channels needed (for LDDs that managed multiple interfaces) and its end of the default endpoint’s pipe channel.

The final result will be a *udi_channel_t* for each USB interface managed by the LDD and a *udi_channel_t* for the default endpoint’s pipe channel. The LDD may then use each interface channel to open the interface or may use the pipe channel to read descriptors from the device[6].

4.7.3 Setting the Configuration

LDDs are not required to, and in most cases need not, perform a *usbdi_config_set_req()* to set the

[6] For a detailed example of the USB interface binding mechanics refer to the sample driver in “Appendix B: usbdi_printer.c” of this specification.

device's configuration. The USBD will set the device's configuration before the LDDs are bound. There may be some LDDs that do need to perform a *usbdi_config_set_req()* (such as some vendor unique LDDs) but in most cases it is not necessary.

4.7.4 Reading Descriptors

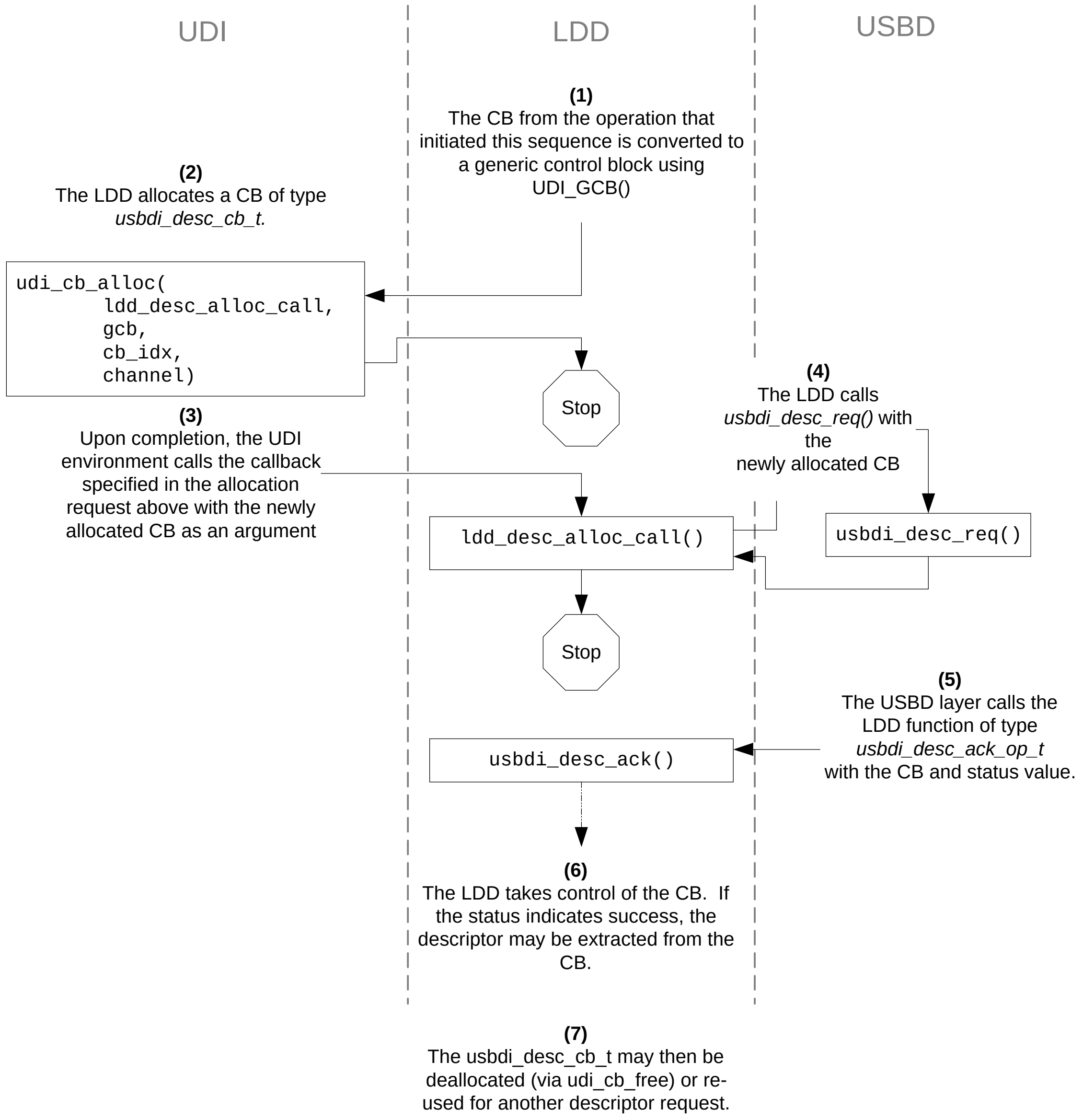
UDI
LDD
USBD
(1)
The CB from the operation that initiated this sequence is converted to a generic control block using UDI_GCB()
(2)
The LDD allocates a CB of type *usbdi_desc_cb_t.*
udi_cb_alloc(
ldd_desc_alloc_call,
gcb,
cb_idx,
channel)
Stop
(3)
Upon completion, the UDI environment calls the callback specified in the allocation request above with the newly allocated CB as an argument
(4)
The LDD calls *usbdi_desc_req()* with the newly allocated CB
ldd_desc_alloc_call()
usbdi_desc_req()
Stop
(5)
The USBD layer calls the LDD function of type *usbdi_desc_ack_op_t* with the CB and status value.
usbdi_desc_ack()
(6)
The LDD takes control of the CB. If the status indicates success, the descriptor may be extracted from the CB.
(7)
The usbdi_desc_cb_t may then be deallocated (via udi_cb_free) or re-used for another descriptor request.

Figure 7: Reading Descriptors

LDDs may retrieve any descriptor provided by the device, including those returned as part of the device's configuration descriptor, by using the *usbdi_desc_req()* function and control block. The LDD does not need to open the interface before calling *usbdi_desc_req()*. Figure 7 shows the functional flow of a descriptor being retrieved.

4.7.5 Opening an Interface

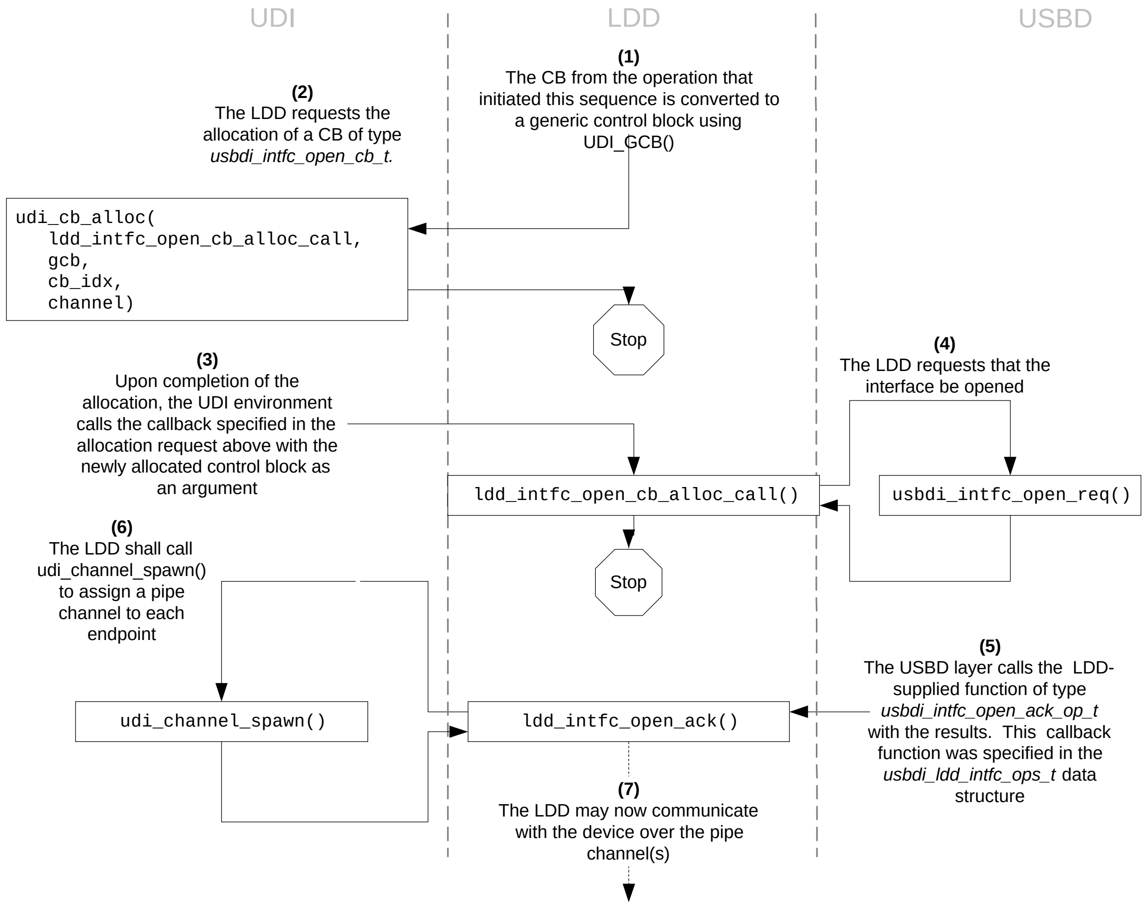
UDI
LDD
USBD
(1)
The CB from the operation that initiated this sequence is converted to a generic control block using UDI_GCB()
(2)
The LDD requests the allocation of a CB of type *usbdi_intfc_open_cb_t*.
udi_cb_alloc(ldd_intfc_open_cb_alloc_call, gcb, cb_idx, channel)
Stop
(3)
Upon completion of the allocation, the UDI environment calls the callback specified in the allocation request above with the newly allocated control block as an argument
(4)
The LDD requests that the interface be opened
ldd_intfc_open_cb_alloc_call()
usbdi_intfc_open_req()
Stop
(6)
The LDD shall call udi_channel_spawn() to assign a pipe channel to each endpoint
(5)
The USBD layer calls the LDD-supplied function of type *usbdi_intfc_open_ack_op_t* with the results. This callback function was specified in the *usbdi_ldd_intfc_ops_t* data structure
udi_channel_spawn()
ldd_intfc_open_ack()
(7)
The LDD may now communicate with the device over the pipe channel(s)

Figure 8: Opening an Interface

An LDD opens an interface by calling *usbdi_intfc_open_req()*, thus allocating the needed USB bandwidth for all endpoints controlled by the interface.

Before the LDD may communicate with endpoints controlled by the interface, the LDD shall create a pipe channel for each endpoint by calling *udi_channel_spawn()* for each one. The channel between the LDD and an endpoint is the communication pipe for that endpoint. Note that the pipe channel for the default endpoint was already created by the LDD during the interface binding sequence. Figure 8 shows the functional flow of an interface being opened.

4.7.6 Closing an Interface

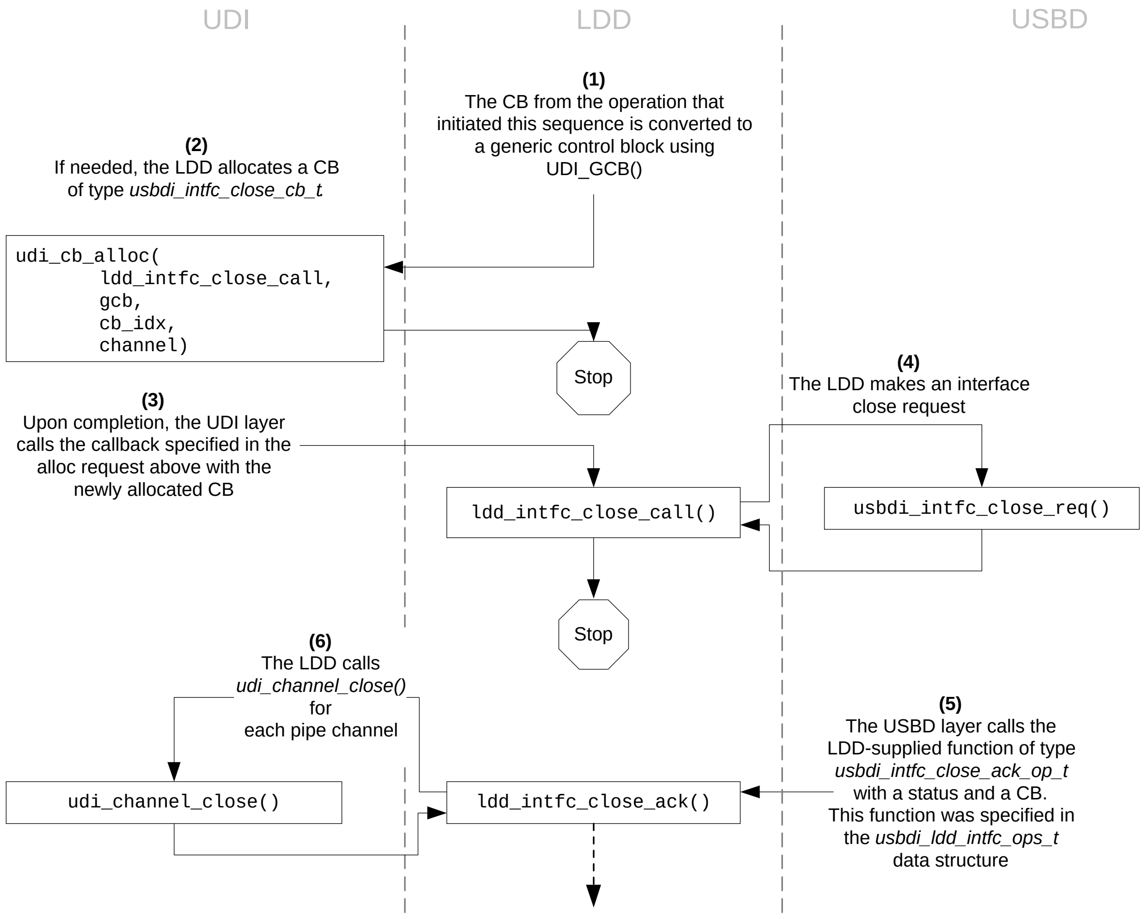
UDI
LDD
USBD
(1)
The CB from the operation that initiated this sequence is converted to a generic control block using UDI_GCB()
(2)
If needed, the LDD allocates a CB of type *usbdi_intfc_close_cb_t*
udi_cb_alloc(ldd_intfc_close_call, gcb, cb_idx, channel)
Stop
(3)
Upon completion, the UDI layer calls the callback specified in the alloc request above with the newly allocated CB
(4)
The LDD makes an interface close request
ldd_intfc_close_call()
usbdi_intfc_close_req()
Stop
(6)
The LDD calls *udi_channel_close()* for each pipe channel
(5)
The USBD layer calls the LDD-supplied function of type *usbdi_intfc_close_ack_op_t* with a status and a CB. This function was specified in the *usbdi_ldd_intfc_ops_t* data structure
udi_channel_close()
ldd_intfc_close_ack()

Figure 9: Closing an Interface

The LDD closes endpoint pipes (channels) by calling *udi_channel_close()* and closes interface channels by calling *usbdi_intfc_close_req()*. Figure 9 shows the flow of an interface being closed. Once the interface has been closed, USB bandwidth that had been used by the pipes will be made available to other interfaces.

After the LDD calls *usbdi_intfc_close_req()* the LDD may receive one or more UDI channel event indications (UDI_CHANNEL_CLOSED type).

4.7.7 Transferring Data

Once a pipe channel has been established between the LDD and an endpoint the LDD may issue requests to the endpoint based on the endpoint type. An endpoint may be of the following types: control, interrupt, bulk, or isochronous.

The following functions are used by the LDD based on the endpoint type: interrupt and bulk endpoints use *usbdi_intr_bulk_xfer_req()*; control endpoints use *usbdi_control_xfer_req()*; and isochronous endpoints use *usbdi_isoc_xfer_req()*.

Although the functions used to transfer data differ depending on the endpoint type, the procedure is the same.

1. The LDD sets up a control block describing the data to be transferred. The type of control block is specific to the endpoint type. Only CBs for control endpoints contain a data transfer direction. The transfer direction is implied by the endpoint direction for all other endpoint types. Control blocks may be reused by LDDs.

2. The LDD calls the appropriate transfer function to initiate the transfer, thus relinquishing its control of the CB and the associated data.

3. The LDD is informed of the request's completion status via a completion routine supplied by the LDD and specific to the control block type. There are two operation routines for each "xfer_req()" function, one for completion with error and one for completion without error. Upon receipt of the control block the LDD regains control of the CB.

Figure 10 shows the code flow for a data transfer between the LDD and the USB device's endpoint.

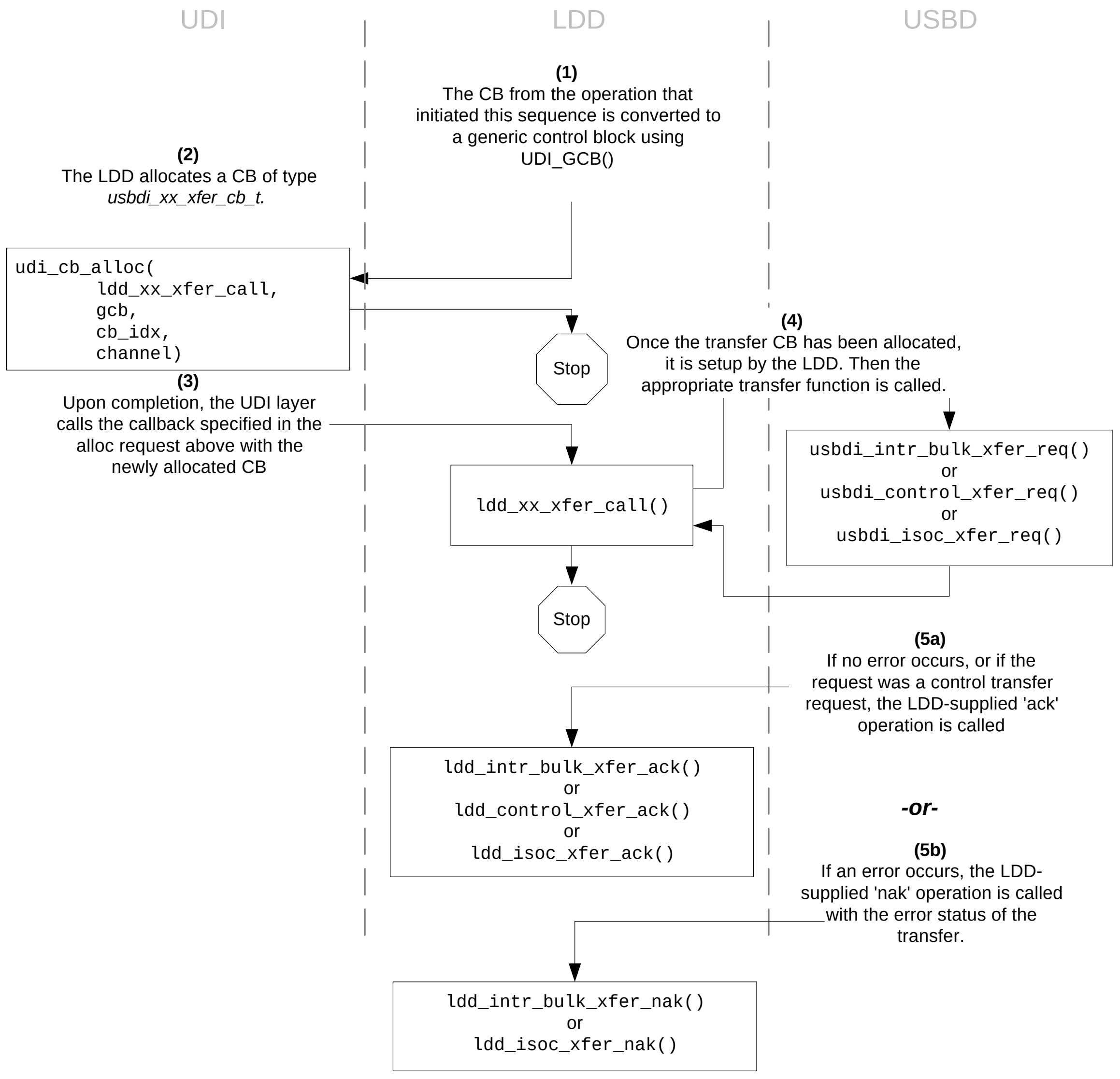
UDI
LDD
USBD
(1)
The CB from the operation that initiated this sequence is converted to a generic control block using UDI_GCB()
(2)
The LDD allocates a CB of type *usbdi_xx_xfer_cb_t*.
udi_cb_alloc(ldd_xx_xfer_call, gcb, cb_idx, channel)
Stop
(3)
Upon completion, the UDI layer calls the callback specified in the alloc request above with the newly allocated CB
(4)
Once the transfer CB has been allocated, it is setup by the LDD. Then the appropriate transfer function is called.
ldd_xx_xfer_call()
usbdi_intr_bulk_xfer_req() or usbdi_control_xfer_req() or usbdi_isoc_xfer_req()
Stop
(5a)
If no error occurs, or if the request was a control transfer request, the LDD-supplied 'ack' operation is called
ldd_intr_bulk_xfer_ack() or ldd_control_xfer_ack() or ldd_isoc_xfer_ack()
-or-
(5b)
If an error occurs, the LDD-supplied 'nak' operation is called with the error status of the transfer.
ldd_intr_bulk_xfer_nak() or ldd_isoc_xfer_nak()

Figure 10: Transferring Data

4.7.8 Completing Transfer Requests

When a transfer operation completes the completion function provided by the LDD for the specific control block type will be called. There are two completion operations, “ack” and “nak”. Completion operation entry points are registered with the USBD as part of the *usbdi_ldd_pipe_ops_t*. (See 4.7.1, “Logical-Device Driver Initialization”).

Once the completion function is called, the LDD has control of the CB and may examine the fields of the structure.

The LDD may reuse the CB or may release it by calling *udi_cb_free()*.

Transfer requests completing with an error shall have one of the following status values:

USBDI_STAT_STALL	The device responded with a USB stall packet.
USBDI_STAT_NOT_ENOUGH_ BANDWIDTH	The request could not be satisfied due to insufficient USB bandwidth. The LDD may wish to resubmit the request later, reformat the request, or abort the request.
UDI_STAT_INVALID_STATE	The pipe or interface is in the USBDI_STATE_IDLE state.
UDI_STAT_MISTAKEN_IDENTITY	The request is understood and implemented, but is inappropriate for the channel or contains invalid values in the CB.
UDI_STAT_ABORTED	The request was successfully aborted as a result of UDI channel abort operation, *usbdi_pipe_abort_req()*, or *usbdi_intfc_abort_req()*.
UDI_STAT_TIMEOUT	The timeout period for the request expired.
UDI_STAT_HW_PROBLEM	A problem has been detected with the associated hardware.
UDI_STAT_NOT_RESPONDING	The device is not responding or is inaccessible. This may be the case if three successive requests to the device were not acknowledged.
UDI_STAT_DATA_UNDERRUN	The device transferred less data than requested.
UDI_STAT_DATA_OVERRUN	The device attempted to transfer more data than requested.
UDI_STAT_DATA_ERROR	An error occurred during the data transfer. The error may have been due to a bit stuffing error, data toggle mismatch, unexpected packet ID, etc.

4.7.9 Pipe State Control

Pipe channels between the LDD and the device's endpoints are created by the LDD as described in Section 4.7.5, "Opening an Interface." When a pipe is first created, its state is initialized to active. In this state, the pipe is ready to accept requests to be queued and sent to the endpoint. If a request completes with a status other than UDI_OK, the state of the pipe will be set by the USBD to USBDI_STATE_STALLED. Pipes support the following states:

USBDI_STATE_ACTIVE	The pipe will accept requests to be queued and will send requests to the endpoint. The endpoint associated with the pipe should be in the USBDI_STATE_ACTIVE state before the pipe is placed in this state.
USBDI_STATE_STALLED	The pipe will accept requests to be queued but will not send requests to the endpoint.
USBDI_STATE_IDLE	The pipe will not accept requests to be queued and will not send requests to the endpoint.

An LDD may set the state of a pipe with *usbdi_pipe_state_set_req()* and may retrieve the state with *usbdi_pipe_state_get_req()*.

Note: The state of the default pipe can't be changed and a *usbdi_pipe_state_set_req()* operation performed on the default pipe channel is considered a no-op.

Note: Setting the pipe state does not affect the state of the endpoint.

4.7.10 Interface State Control

Pipe's state may also be manipulated as an interface group with the *usbdi_intfc_state_set_req()* and *usbdi_intfc_state_get_req()*. When an LDD opens an interface, the interface state is initialized to USBDI_STATE_ACTIVE. An interface may be in one of the following states:

USBDI_STATE_ACTIVE	When used by *usbdi_intfc_state_set_req()* the interface and all associated pipes will be moved to the active state regardless of the prior state of any of the pipes[7]. When the USBDI_STATE_ACTIVE state is returned by the *usbdi_intfc_state_get_req()* function, the interface is active although pipes controlled by the interface may not be. The state of individual pipes (except for the default endpoint pipe) may be changed by *usbdi_pipe_state_set_req()*.
USBDI_STATE_STALLED	All associated pipes are in the USBDI_STATE_STALLED state. A pipe's state may not be modified by *usbdi_pipe_state_set_req()* while the interface is in the USBDI_STATE_STALLED state. All pipes controlled by the interface shall be manipulated as an interface group while the interface state is USBDI_STATE_STALLED.
USBDI_STATE_IDLE	All associated pipes are in the USBDI_STATE_IDLE state. A pipe's state may not be modified by *usbdi_pipe_state_set_req()* while the state of the interface is USBDI_STATE_IDLE. All pipes contained by the interface shall be manipulated as an interface group while the interface state is USBDI_STATE_IDLE.

Note: The state of the default endpoint pipe cannot be changed and is unaffected by *usbdi_intfc_state_set_req()*.

[7] Note that it is the responsibility of the LDD to handle any endpoint stall conditions and clear them as needed by calling *usbdi_edpt_state_set_req()*.

4.7.11 Endpoint State Control

When a device receives a request that either is not defined for the device, is inappropriate for the current setting of the device, or has values that are not compatible with the request, the device returns a STALL packet identifier[8]. The LDD shall perform a *usbdi_edpt_state_set_req()* with a status of USBDI_STATE_ACTIVE to clear the device's "endpoint halted" condition and to reset the host controller's data toggle after resolving the condition that caused the endpoint stall. See the Universal Serial Bus Specification for more information on STALL, "endpoint halted", and data toggle.

4.7.12 Aborting Transfer Requests

During error recovery, an LDD may choose to abort requests that have been queued to a pipe. This may be accomplished with the following functions:

udi_channel_op_abort()	The transfer control block identified by the *orig_cb* given as an argument to *udi_channel_op_abort()* will be aborted and returned to the LDD's completion function for that control block type with a status of UDI_STAT_ABORTED. The pipe may be in any state when this function is called and will be in the USBDI_STATE_STALLED state when this function completes. This function shall not be called if the transfer control block identified by *orig_cb* has completed by being returned via the approriate request completion call.
usbdi_pipe_abort_req()	All transfer control blocks queued to the pipe will be returned to the LDD's completion function for the transfer control block with a status of UDI_STAT_ABORTED. The pipe may be in any state when this function is called, and will be in the USBDI_STATE_STALLED state when the pipe abort process completes.
usbdi_intfc_abort_req()	All transfer control blocks queued to all pipes of the interface are returned to the LDD's completion function for the transfer control block with a status of UDI_STAT_ABORTED. The interface may be in any state when this function is called, and will be in the USBDI_STATE_STALLED state when the abort process completes.

The LDD shall clear stall conditions (see the Universal Serial Bus Core Specification) on USB endpoints by calling *usbdi_edpt_state_set_req()*. It can obtain the state of the endpoint by calling *usbdi_edpt_state_get_req()*.

[8] Refer to the Universal Serial Bus Specification for the meaning of this term.

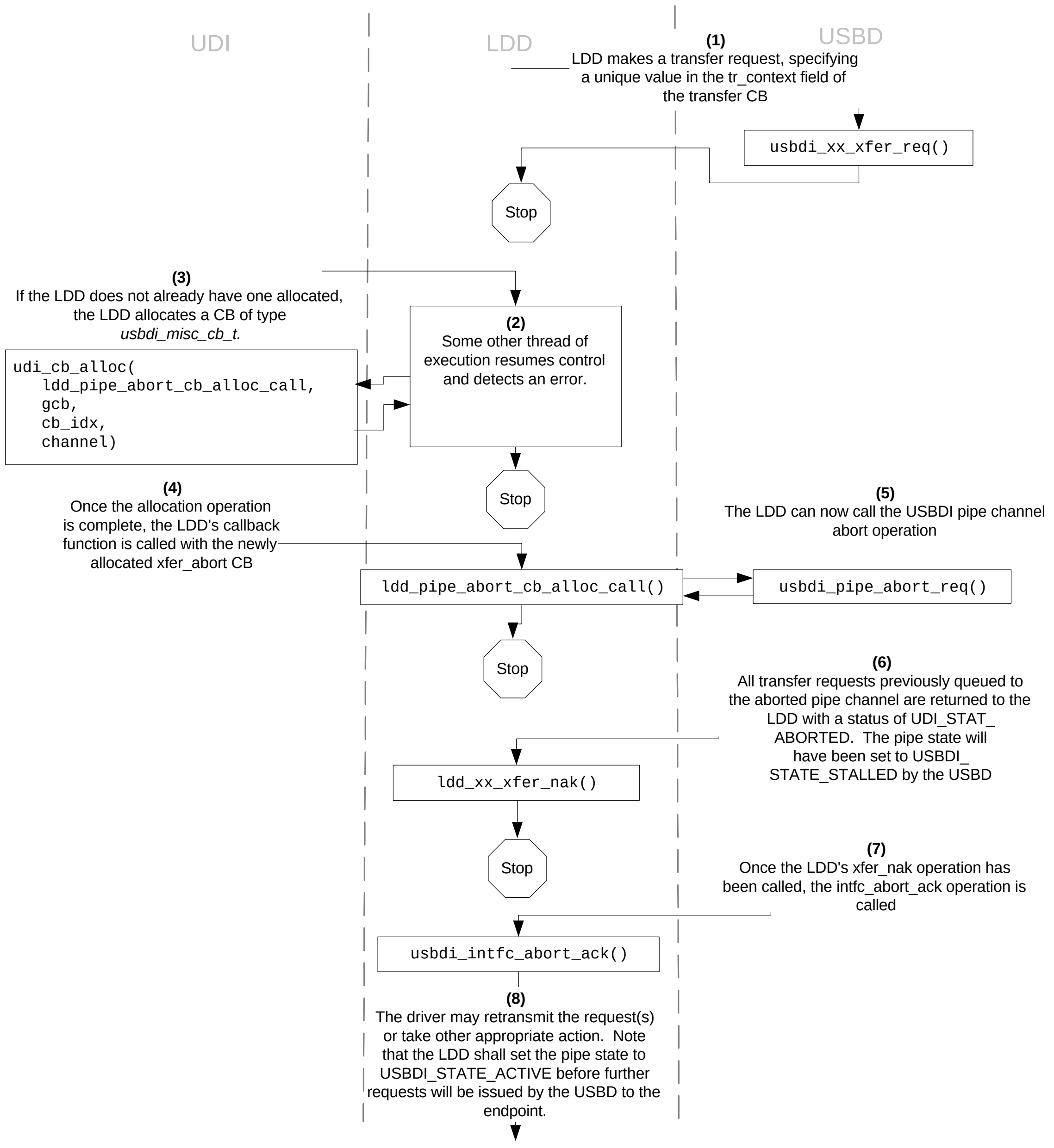
UDI
LDD
USBD
(1)
LDD makes a transfer request, specifying a unique value in the tr_context field of the transfer CB
usbdi_xx_xfer_req()
Stop
(3)
If the LDD does not already have one allocated, the LDD allocates a CB of type *usbdi_misc_cb_t*.
udi_cb_alloc(ldd_pipe_abort_cb_alloc_call, gcb, cb_idx, channel)
(2)
Some other thread of execution resumes control and detects an error.
Stop
(4)
Once the allocation operation is complete, the LDD's callback function is called with the newly allocated xfer_abort CB
(5)
The LDD can now call the USBDI pipe channel abort operation
ldd_pipe_abort_cb_alloc_call()
usbdi_pipe_abort_req()
Stop
(6)
All transfer requests previously queued to the aborted pipe channel are returned to the LDD with a status of UDI_STAT_ ABORTED. The pipe state will have been set to USBDI_ STATE_STALLED by the USBD
ldd_xx_xfer_nak()
Stop
(7)
Once the LDD's xfer_nak operation has been called, the intfc_abort_ack operation is called
usbdi_intfc_abort_ack()
(8)
The driver may retransmit the request(s) or take other appropriate action. Note that the LDD shall set the pipe state to USBDI_STATE_ACTIVE before further requests will be issued by the USBD to the endpoint.

Figure 11: Aborting a pipe

Figure 11 shows the functional flow involved in aborting all transfers on a USB pipe during error recovery.

5 OpenUSBDI Metalanguage

The control block functions and function *typedef's* described in this section define how the LDD and the USBD communicate. Many of the OpenUSBDI operation functions use the same control blocks such as the *usbdi_misc_cb_t* and the *usbdi_state_cb_t*. Other control blocks are specific for particular operations. The following naming scheme is used for OpenUSBDI control blocks and operation functions:

usbdi_XX_req()	Called by the LDD to send the CB to the USBD.
usbdi_XX_req_op_t	Function type for the USBD supplied function that will be called in response to the LDD performing the *usbdi_XX_req()* call.
usbdi_XX_ack()	Called by the USBD on completion of the operation.
usbdi_XX_ack_op_t	Function type for the LDD supplied function that will be called on completion of the operation.
usbdi_XX_nak()	Called by the USBD when an operation completes with an error. Only used for "xfer" operations.
usbdi_XX_nak_op_t	Function type for the LDD supplied function that will be called when an operation completes with an error. Only used for "xfer" operations.

To perform a request, the LDD allocates a control block by calling *udi_cb_alloc()* with a CB-specific index argument and a UDI channel. The index argument is the same index value that was given to *usbdi_XX_cb_init()* at driver initialization time. The UDI channel may be a bind channel, an interface channel, or a pipe channel depending on the operation being performed.

The LDD sets up this CB and sends it to the USBD by calling *usbdi_XX_req()*. The LDD shall provide a *usbdi_XX_ack_op_t* function that will be called when the operation completes.

usbdi_XX_req_op_t , *usbdi_XX_ack()*, and *usbdi_XX_nak()*, are provided for completeness. They are used only by the USBD (not by the LDD) and are provided in **open_usbdi.h** (Appendix A: Include File (open_usbdi.h)), but not described in the following sections. Their parameters are identical to the corresponding LDD function prototypes.

5.1 Driver Initialization Structures

The UDI Initialization Core Services specify that each UDI driver shall have a global symbol named `udi_init_info` of type `udi_init_t`. This structure contains information describing the module's entry points, control block usage, and other information necessary to initialize the driver. The environment processes the information contained in this structure before executing any driver code.

Among the data structures referenced by each `udi_init_info` is a list of metalanguage operation vector structures. There are two such structures defined by the OpenUSBDI metalanguage: `usbdi_ldd_intfc_ops_t` and `usbdi_ldd_pipe_ops_t`.

All LDDs shall register their interface and pipe operation functions defined by *usbdi_ldd_intfc_ops_t* and *usbdi_ldd_pipe_ops_t* by referencing them in a *udi_ops_init_t* structure that is referenced by the LDD's *udi_init_info*.

This section contains descriptions for the following *typedefs*:

- *usbdi_ldd_intfc_ops_t*
- *usbdi_ldd_pipe_ops_t*

5.1.1 usbdi_ldd_intfc_ops_t

Name

`usbdi_ldd_intfc_ops_t`

Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

/*
 * Channel operation entry points for LDD side of bind channels
 * and USB interface channels.
 */
typedef const struct {
    udi_channel_event_ind_op_t      *udi_channel_event_ind_op;
    usbdi_bind_ack_op_t             *usbdi_bind_ack_op;
    usbdi_unbind_ack_op_t           *usbdi_unbind_ack_op;
    usbdi_intfc_open_ack_op_t       *usbdi_intfc_open_ack_op;
    usbdi_intfc_close_ack_op_t      *usbdi_intfc_close_ack_op;
    usbdi_frame_number_ack_op_t     *usbdi_frame_number_ack_op;
    usbdi_device_speed_ack_op_t     *usbdi_device_speed_ack_op;
    usbdi_reset_device_ack_op_t     *usbdi_reset_device_ack_op;
    usbdi_intfc_abort_ack_op_t      *usbdi_intfc_abort_ack_op;
    usbdi_intfc_state_set_ack_op_t  *usbdi_intfc_state_set_ack_op;
    usbdi_intfc_state_get_ack_op_t  *usbdi_intfc_state_get_ack_op;
    usbdi_desc_ack_op_t             *usbdi_desc_ack_op;
    usbdi_device_state_get_ack_op_t *usbdi_device_state_get_ack_op;
    usbdi_config_set_ack_op_t       *usbdi_config_set_ack_op;
    usbdi_async_event_ind_op_t      *usbdi_async_event_ind_op;
} usbdi_ldd_intfc_ops_t;

#define USBDI_LDD_INTFC_OPS_NUM  1
```

Description

A USB LDD uses the `usbdi_ldd_intfc_ops_t` structure in a `udi_ops_init_t` as part of its `udi_init_info` in order to register its interface-specific OpenUSBDI Metalanguage entry points.

5.1.2 usbdi_ldd_pipe_ops_t

Name

```
usbdi_ldd_pipe_ops_t
```

Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

/*
 * Channel operation entry points for LDD side of pipe channels.
 */
typedef const struct {
    udi_channel_event_ind_op_t    *udi_channel_event_ind_op;
    usbdi_intr_bulk_xfer_ack_op_t *usbdi_intr_bulk_xfer_ack_op;
    usbdi_intr_bulk_xfer_nak_op_t *usbdi_intr_bulk_xfer_nak_op;
    usbdi_control_xfer_ack_op_t   *usbdi_control_xfer_ack_op;
    usbdi_isoc_xfer_ack_op_t      *usbdi_isoc_xfer_ack_op;
    usbdi_isoc_xfer_nak_op_t      *usbdi_isoc_xfer_nak_op;
    usbdi_pipe_abort_ack_op_t     *usbdi_pipe_abort_ack_op;
    usbdi_pipe_state_set_ack_op_t *usbdi_pipe_state_set_ack_op;
    usbdi_pipe_state_get_ack_op_t *usbdi_pipe_state_get_ack_op;
    usbdi_edpt_state_set_ack_op_t *usbdi_edpt_state_set_ack_op;
    usbdi_edpt_state_get_ack_op_t *usbdi_edpt_state_get_ack_op;
} usbdi_ldd_pipe_ops_t;

#define USBDI_LDD_PIPE_OPS_NUM   2
```

Description

A USB LDD uses the `usbdi_ldd_pipe_ops_t` structure in a `udi_ops_init_t` as part of its `udi_init_info` in order to register its pipe-specific OpenUSBDI Metalanguage entry points.

5.2 Miscellaneous Control Block

The miscellaneous control block is used by many of the OpenUSBDI operations.This section contains descriptions for the following function and *typedef*:

- *usbdi_misc_cb_t*

5.2.1 usbdi_misc_cb_t

***usbdi_misc_cb_t* – Used in various OpenUSBDI channel operations.**

5.2.1.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

typedef struct {

      udi_cb_t gcb;

} usbdi_misc_cb_t;

#define USBDI_MISC_CB_NUM     1
```

5.2.1.2 Description

A USB LDD uses the `usbdi_misc_cb_t` structure in a `udi_cb_init_t` as part of its `udi_init_info` in order to register its use of the miscellaneous OpenUSBDI Metalanguage control block.

In order to use this type of control block it must be associated with a control block index by including `USBDI_MISC_CB_NUM` in a `udi_cb_init_t` in the driver's `udi_init_info`.

5.3 Driver Binding

This group of functions allow an LDD to be bound to a USB logical device.

This section contains descriptions for the following function and *typedef*:

- *usbdi_bind_req()*
- *usbdi_bind_ack_op_t*

The following function and *typedef* are used only by the USBD (not by the LDD) and are not described in this specification. Callee-side interfaces are identical to the corresponding caller-side interfaces. They are included in "Appendix A: Include File (open_usbdi.h)" for completeness:

- *usbdi_bind_req_op_t*
- *usbdi_bind_ack()*

5.3.1 usbdi_bind_req()

***usbdi_bind_req()* – Called by the LDD to bind a UDI channel to a USB logical-device.**

5.3.1.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

void usbdi_bind_req(
            usbdi_misc_cb_t *cb);
```

5.3.1.2 Arguments

usbdi_misc_cb_t ****cb***	Pointer to control block that was allocated with *udi_cb_alloc()* given a cb_idx that was previously associated with USBDI_MISC_CB_NUM and given the LDD's bind channel.

5.3.1.3 Description

usbdi_bind_req() is called by the LDD to bind UDI channel(s) to a USB logical-device. This function shall be called before any requests may be issued to the logical-device. The bind process is complete when the LDD's *usbdi_bind_ack_op_t* function is called.

5.3.2 usbdi_bind_ack_op_t

***usbdi_bind_ack_op_t* – *typedef* for LDD supplied function called by the USBD in response to a *usbdi_bind_req()* call.**

5.3.2.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

typedef
void usbdi_bind_ack_op_t(
            usbdi_misc_cb_t *cb,
            udi_index_t n_intfc,
            udi_status_t status);
```

5.3.2.2 Arguments

usbdi_misc_cb_t ****cb***	Pointer to control block that was passed to *usbdi_bind_req()*.
udi_index_t ***n_intfc***	Number of interfaces that the LDD is in control of. Each interface shall be bound before it may be used.
udi_status_t ***status***	See table below.

RETURNED STATUS	**DESCRIPTION**
UDI_OK	Bind request completed properly.
UDI_STAT_INVALID_STATE	A channel for the logical-device has already been bound.

5.3.2.3 Description

The LDD shall supply a *usbdi_bind_ack_op_t* function in its *usbdi_ldd_intfc_ops_t*. This function is called in response to the LDD calling *usbdi_bind_req()*. Once the LDD's *bind_ack()* function is called with a status of UDI_OK, the LDD shall spawn its end of the default endpoint channel. The LDD may then use the USB interface channel that was just bound. If there is more than one interface (***n_intfc*** is greater than one), the LDD shall also spawn its end of all additional interface channels. Spawn index zero ("0") shall be used for the default endpoint's channel; spawn index one ("1") shall be used for the first additional interface channel; spawn index two ("2") for the next one; and so on.

5.4 Driver Unbinding

This group of functions allow an LDD to be unbound from a USB logical-device.

This section contains descriptions for the following function and *typedef*:

- *usbdi_unbind_req()*
- *usbdi_unbind_ack_op_t*

The following function and *typedef* are used only by the USBD (not by the LDD) and are not described in this specification. Callee-side interfaces are identical to the corresponding caller-side interfaces. They are included in "Appendix A: Include File (open_usbdi.h)" for completeness:

- *usbdi_unbind_req_op_t*
- *usbdi_unbind_ack()*

5.4.1 usbdi_unbind_req()

***usbdi_unbind_req()* – Called by the LDD to unbind a UDI channel from a USB logical-device.**

5.4.1.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

void usbdi_unbind_req(
            usbdi_misc_cb_t *cb);
```

5.4.1.2 Arguments

usbdi_misc_cb_t ****cb***	Pointer to control block that was allocated with *udi_cb_alloc()* given a cb_idx that was previously associated with USBDI_MISC_CB_NUM and given the LDD's bind channel.

5.4.1.3 Description

usbdi_unbind_req() is called by the LDD to unbind a UDI channel from a USB logical-device. All pending requests must be completed and no new requests may be issued to the logical-device once this function is called. The unbind process is complete when the LDD's *usbdi_unbind_ack_op_t* function is called.

The LDD is responsible for closing its end of the default pipe channel and all interface channels other than the first one (the bind channel).

5.4.2 usbdi_unbind_ack_op_t

***usbdi_unbind_ack_op_t* – *typedef* for LDD supplied function called by the USBD in response to a *usbdi_unbind_req()* call.**

5.4.2.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

typedef
void usbdi_unbind_ack_op_t(
            usbdi_misc_cb_t *cb,
            udi_status_t status);
```

5.4.2.2 Arguments

usbdi_misc_cb_t ****cb***	Pointer to control block that was passed to *usbdi_unbind_req()*.
udi_status_t ***status***	See table below.

RETURNED STATUS	**DESCRIPTION**
UDI_OK	Unbind request completed properly.
UDI_STAT_INVALID_STATE	An interface channel has not been closed.

5.4.2.3 Description

The LDD shall supply a *usbdi_unbind_ack_op_t* function in its *usbdi_ldd_intfc_ops_t*. This function is called in response to the LDD calling *usbdi_unbind_req()*.

All interface channels controlled by the LDD shall be closed before a *usbdi_unbind_req()* is performed (See Section 5.6, “Closing an Interface”).

The LDD is responsible for closing its end of the default pipe channel and all interface channels other than the first one (the bind channel).

5.5 Opening an Interface

Opening an interface results in the allocation of bandwidth for all pipes in the interface by the USBD. The LDD may open the default interface or a valid alternate interface using *usbdi_intfc_open_req()*. If the interface being opened is the default interface, *alternate_intfc* shall be set to zero. If an alternate interface is being opened, *alternate_intfc* shall be set to the alternate interface number, as found in the *usb_interface_descriptor_t*, for the interface that has been bound to the LDD.

This section contains descriptions for the following function and *typedef*:

- *usbdi_intfc_open_req()*
- *usbdi_intfc_open_ack_op_t*

The following function and *typedef* are used only by the USBD (not by the LDD). Callee-side interfaces are identical to the corresponding caller-side interfaces. They are not described in this specification, but are included in “Appendix A: Include File (open_usbdi.h)” for completeness:

- *usbdi_intfc_open_req_op_t*
- *usbdi_intfc_open_ack()*

5.5.1 usbdi_intfc_open_req()

***usbdi_intfc_open_req()* – Called by the LDD to open a USB interface.**

5.5.1.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

void usbdi_intfc_open_req(
            usbdi_misc_cb_t *cb,
            udi_ubit8_t alternate_intfc,
            udi_ubit8_t open_flag );
```

5.5.1.2 Arguments

usbdi_misc_cb_t ***cb***	Pointer to control block that was allocate by *udi_cb_alloc()* given a cb_idx that was previously associated with USBDI_MISC_CB_NUM and given a LDD's interface channel.
udi_ubit8_t ***alternate_intfc***	Alternate interface number to open, zero if default interface.
udi_ubit8_t ***open_flag***	Flags providing additional details for the interface open request. See table below.

OPEN_FLAG VALUES	**DESCRIPTION**
USBDI_INTFC_OPEN_ASYNC_EVENT	The LDD is prepared to accept asynchronous notifications as described in Section 5.22.3 "Asynchronous Events".

5.5.1.3 Description

usbdi_intfc_open_req() is called by the LDD to allocate USB bandwidth for the interface and to open the USBD end of the pipe channels. This function shall be called before the LDD may bind its end of the pipe channels for the USB endpoints controlled by the interface.

5.5.2 usbdi_intfc_open_ack_op_t

***usbdi_intfc_open_ack_op_t – typedef* for LDD supplied function called by the USBD on completion of an open interface operation**.

5.5.2.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

typedef
void usbdi_intfc_open_ack_op_t(
            usbdi_misc_cb_t *cb,
            udi_index_t n_edpt,
            udi_status_t status );
```

5.5.2.2 Arguments

usbdi_misc_cb_t ****cb***	Pointer to control block that was passed to *usbdi_intfc_open_req()*.
udi_index_t ***n_edpt***	Number of endpoints (not including the default endpoint) that are associated with this interface.
udi_status_t ***status***	See table below.

RETURNED STATUS	**DESCRIPTION**
UDI_OK	Interface open request completed properly.
USBDI_STAT_NOT_ENOUGH_BANDWIDTH	There was not enough available bandwidth on the USB to open this interface.
UDI_STAT_NOT_RESPONDING	The device is not responding or is inaccessible.
UDI_STAT_MISTAKEN_IDENTITY	The alternate_intfc value given to usbdi_intfc_open_req() was invalid.
UDI_STAT_INVALID_STATE	The interface was already opened.

5.5.2.3 Description

The LDD shall supply a *usbdi_intfc_open_ack_op_t* function in its *usbdi_ldd_intfc_ops_t*. This function is called in response to the LDD's calling *usbdi_intfc_open_req()*. Once the LDD's *intfc_open_ack_op_t* operation is called with a status of UDI_OK, the LDD may proceed with spawning its end of channels (creating pipes) for each endpoint controlled by the interface by calling *udi_channel_spawn()*[9]. Note that the LDD already spawned a channel for the default interface when the driver was bound (See section 5.3.2, *usbdi_bind_ack_op_t*). The spawn index for the first endpoint is one ("1"); the spawn index for the second endpoint is two ("2"); and so on.

The LDD shall either free the *usbdi_misc_cb_t* by calling *udi_cb_free()* or reuse the CB.

[9] See sample driver in Appendix B for an example.

5.6 Closing an Interface

Closing a USB interface results in all bandwidth for the interface being released and made available to other USB interfaces.

This section contains descriptions for the following function and *typedef*:

- *usbdi_intfc_close_req*
- *usbdi_intfc_close_ack_op_t*

The following function and *typedef* are used only by the USBD (not by the LDD). Callee-side interfaces may be derived from the caller-side interfaces by removing the first parameter, and vice versa. They are not described in this specification but are included in "Appendix A: Include File (open_usbdi.h)" for completeness:

- *usbdi_intfc_close_req_op_t*
- *usbdi_intfc_close_ack()*

5.6.1 usbdi_intfc_close_req()

***usbdi_intfc_close_req()* – Called by the LDD to close a USB interface.**

5.6.1.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

void usbdi_intfc_close_req(
            usbdi_misc_cb_t *cb );
```

5.6.1.2 Arguments

usbdi_misc_cb_t ***cb***	Pointer to control block that was allocated by *udi_cb_alloc()* given a cb_idx that was previously associated with USBDI_MISC_CB_NUM and given a LDD's interface channel.

5.6.1.3 Description

usbdi_intfc_close_req() is called by the LDD to request that the USBD release all USB bandwidth reserved for the interface and to close the USBD end of the pipe channels. Once the LDD calls *usbdi_intfc_close_req()* it may receive UDI channel event indications (UDI_CHANNEL_CLOSED) for the pipe channels.

The LDD shall not have outstanding requests to any pipe channels and shall not perform pipe channel operations once *usbdi_intfc_close_req()* has been called.

5.6.2 usbdi_intfc_close_ack_op_t

***usbdi_intfc_close_ack_op_t – typedef* for LDD supplied function called by the USBD on the completion of the close of a USB interface operation.**

5.6.2.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

typedef
void usbdi_intfc_close_ack_op_t(
            usbdi_misc_cb_t *cb,
            udi_status_t status );
```

5.6.2.2 Arguments

usbdi_misc_cb_t ****cb***	Pointer to control block that was passed to *usbdi_intfc_close_req()*.
udi_status_t ***status***	See table below.

RETURNED STATUS	**DESCRIPTION**
UDI_OK	Interface close request completed properly.
UDI_STAT_INVALID_STATE	Interface has already been closed.

5.6.2.3 Description

The LDD shall supply a *usbdi_intfc_close_ack_op_t* function in its *usbdi_ldd_intfc_ops_t*. This function is called in response to the LDD's calling *usbdi_intfc_close_req()*. The interface is closed if ***status*** is UDI_OK.

The LDD is responsible for closing its end of the pipe channels after *usbdi_intfc_close_ack_op_t* is called by calling *udi_channel_close()* for each channel.

The LDD shall either free the *usbdi_misc_cb_t* by calling *udi_cb_free()* or reuse the CB.

5.7 USB Interrupt and Bulk Transfer Requests

This section contains descriptions for the following control block, functions, and *typedefs*:

- *usbdi_intr_bulk_xfer_cb_t*
- *usbdi_intr_bulk_xfer_req()*
- *usbdi_intr_bulk_xfer_ack_op_t*
- *usbdi_intr_bulk_xfer_nak_op_t*
- *usbdi_intr_bulk_xfer_ack_unused()*
- *usbdi_intr_bulk_xfer_nak_unused()*

The following functions and *typedef* are used only by the USBD (not by the LDD). Callee-side interfaces may be derived from the caller-side interfaces by removing the first parameter, and vice versa. They are not described in this specification but are included in “Appendix A: Include File (open_usbdi.h)” for completeness:

- *usbdi_intr_bulk_xfer_req_op_t*
- *usbdi_intr_bulk_xfer_ack()*
- *usbdi_intr_bulk_xfer_nak()*

Bulk endpoints are not used by low speed USB devices. See the Universal Serial Bus Specification for more information.

5.7.1 usbdi_intr_bulk_xfer_cb_t

***usbdi_intr_bulk_xfer_cb_t* – Used in transferring data to or from an interrupt or bulk endpoint.**

5.7.1.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>


typedef struct {

      udi_cb_t gcb;         /* UDI generic control block */

      /*
       * data_buf is set to refer to the buffer where the data
       * will be transferred to/from.  The LDD sets the
       * data_buf->buf_size field to the maximum number of bytes that
       * may be transferred.  To specify a zero length data transfer
       * the data_buf->buf_size shall be set by the LDD to zero.
       *
       * On completion, the USBD sets the data_buf->buf_size
       * with the actual number of bytes transferred.
       */
      udi_buf_t *data_buf;

      /*
       * Request timeout value in milliseconds.  Request timeout
       * periods begin when the USBD receives the transfer control
       * block and NOT when the host controller issues the
       * request to/from the device.  A timeout value of zero
       * specifies an infinite timeout period.  If a request times
       * out, it will be completed by the USBD with a status of
       * UDI_STAT_TIMEOUT.  The associated pipe will be left in the
       * USBDI_STATE_STALLED state.
       */
      udi_ubit32_t timeout;

      /*
       * Flags specific to this request.  The valid flags for
       * interrupt and bulk requests are USBDI_XFER_IN, USBDI_XFER_OUT,
       * and USBDI_XFER_SHORT_OK
       * USBDI_XFER_SHORT_OK specifies a final transfer count that
       * is less than data_buf->buf_size will NOT generate an error
       * and will NOT stall the pipe.
       */
      udi_ubit8_t xfer_flags;
#define USBDI_XFER_SHORT_OK (1<<0)
#define USBDI_XFER_IN       (1<<2)/* transfer data from the device */
#define USBDI_XFER_OUT      (1<<3)/* transfer data to the device */

} usbdi_intr_bulk_xfer_cb_t;
```

```
#define USBDI_INTR_BULK_XFER_CB_NUM       2
```

5.7.1.2 Description

The interrupt and bulk transfer control block is used in OpenUSBDI operations that involve transferring data to and from bulk and interrupt endpoints.

In order to use this type of control block it must be associated with a control block index by including `USBDI_INTR_BULK_XFER_CB_NUM` in a `udi_cb_init_t` in the driver's `udi_init_info`.

5.7.2 usbdi_intr_bulk_xfer_req()

usbdi_intr_bulk_xfer_req() **– Called by the LDD to initiate a transfer with an interrupt or bulk endpoint.**

5.7.2.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

void usbdi_intr_bulk_xfer_req(
            usbdi_intr_bulk_xfer_cb_t *cb );
```

5.7.2.2 Arguments

usbdi_intr_bulk_xfer_cb_t ****cb***	Pointer to control block that was allocated by *udi_cb_alloc()* given a cb_idx that was previously associated with USBDI_INTR_BULK_XFER_CB_NUM and given a LDD's pipe channel.

5.7.2.3 Description

usbdi_intr_bulk_xfer_req() is called by the LDD to transfer data with an interrupt or bulk endpoint.

The *data_buf->buf_size* field of the CB specifies the number of bytes to transfer. It shall be set to zero if no data is to be transferred. It shall be set to a positive, non-zero value if data is to be transferred.

The *data_buf* field is set to a handle for the buffer that contains or will contain the data. For output, *data_buf* shall contain *data_buf->buf_size* bytes of valid data. For input, *data_buf* shall contain zero ("0") bytes of valid data.

The *timeout* field specifies the number of milliseconds that may pass from the time the CB is received by the USBD until the time the CB is passed to *usbdi_intr_bulk_xfer_ack_op_t*. This value may be zero ("0") if an infinite timeout period is needed. In this case, the USBD will never time out the request, although the CB may still be aborted by the LDD.

The *xfer_flags* field of the CB shall be set to zero ("0"), USBDI_XFER_SHORT_OK, USBDI_XFER_IN, and / or USBDI_XFER_OUT. USBDI_XFER_IN specifies that data will be transferred from the device to *data_buf*. USBDI_XFER_OUT specifies that data will be transferred from *data_buf* to the device. If *data_buf->buf_size* is greater than zero, USBDI_XFER_IN or USBDI_XFER_OUT shall be set (only one may be set). If USBDI_XFER_SHORT_OK is set, and the device transfers less than *data_buf->buf_size* bytes, the control block will complete with a status of UDI_OK. See Section 5.7.3, *usbdi_intr_bulk_xfer_ack_op_t*, for more information.

The USBD may queue the request on the pipe if the pipe's state is USBDI_STATE_ACTIVE or USBDI_STATE_STALLED. The request will be sent to the endpoint if the pipe's state is

USBDI_STATE_ACTIVE. See Section 5.17.3, *usbdi_pipe_state_get_req()*, for more information.

5.7.3 usbdi_intr_bulk_xfer_ack_op_t

***usbdi_intr_bulk_xfer_ack_op_t – typedef* for LDD supplied function called by the USBD on the completion of an interrupt or bulk transfer operation without an error.**

5.7.3.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

typedef void usbdi_intr_bulk_xfer_ack_op_t(
            usbdi_intr_bulk_xfer_cb_t *cb );
```

5.7.3.2 Arguments

usbdi_intr_bulk_xfer_cb_t ***cb***	Pointer to control block that was passed to *usbdi_intr_bulk_xfer_req().*

5.7.3.3 Description

The LDD shall supply a *usbdi_intr_bulk_xfer_ack_op_t* function in its *usbdi_ldd_pipe_ops_t*. This function is called by the USBD in response to the LDD's calling *usbdi_intr_bulk_xfer_req()* on completion of the operation without an error.

If USBDI_XFER_SHORT_OK was set in the *xfer_flags* of the CB, then *data_buf->buf_size* shall be examined by the LDD to determine the actual number of bytes transferred.

The LDD shall either free the *usbdi_intr_bulk_xfer_cb_t* by calling *udi_cb_free()* or reuse the CB for another call to *usbdi_intr_bulk_xfer_req().*

5.7.4 usbdi_intr_bulk_xfer_nak_op_t

***usbdi_intr_bulk_xfer_nak_op_t – typedef* for LDD supplied function called by the USBD on the completion of an interrupt or bulk transfer operation with an error.**

5.7.4.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

typedef void usbdi_intr_bulk_xfer_nak_op_t(
           usbdi_intr_bulk_xfer_cb_t *cb,
           udi_status_t status );
```

5.7.4.2 Arguments

usbdi_intr_bulk_xfer_cb_t ****cb***	Pointer to control block that was passed to *usbdi_intr_bulk_xfer_req().*
udi_status_t ***status***	See table below.

RETURNED STATUS	DESCRIPTION
USBDI_STAT_STALL	The device responded with a USB stall.
UDI_STAT_ABORTED	The request was successfully aborted as a result of *udi_channel_op_abort(), usbdi_pipe_abort_req()* or *usbdi_intfc_abort_req().*
UDI_STAT_INVALID_STATE	The pipe is in the USBDI_STATE_IDLE state.
UDI_STAT_DATA_OVERRUN	The device attempted to transfer more data than requested.
UDI_STAT_DATA_UNDERRUN	The device transferred less data than requested and USBDI_XFER_SHORT_OK was not set.
UDI_STAT_TIMEOUT	The timeout period for the request expired.
UDI_STAT_NOT_RESPONDING	The device is not responding or is inaccessible.
UDI_STAT_DATA_ERROR	An error occurred during the data transfer. The error may have been due to a bit stuffing error, data toggle mismatch, unexpected packet ID, etc.
UDI_STAT_MISTAKEN_IDENTITY	A field in the CB is invalid.

5.7.4.3 Description

The LDD shall supply a *usbdi_intr_bulk_xfer_nak_op_t* function in its *usbdi_ldd_pipe_ops_t*. This function is called in response to the LDD's calling *usbdi_intr_bulk_xfer_req()* when the operation completes with an error.

The state of the pipe will be set to USBDI_STATE_STALLED before this function is called, unless the pipe was already in the USBDI_STATE_STALLED state when *usbdi_intr_bulk_xfer_req()* was called. See Section 5.17, "Getting and Setting Pipe States", for more information.

The *data_buf* value of CB shall be unchanged by the USBD. The *data_buf* is now owned by the LDD.

The LDD shall either free the *usbdi_intr_bulk_xfer_cb_t* by calling *udi_cb_free()* or reuse the CB for another call to *usbdi_intr_bulk_xfer_req()*.

5.7.5 usbdi_intr_bulk_xfer_ack_unused()

***usbdi_intr_bulk_xfer_ack_unused()* - Proxy for *usbdi_intr_bulk_xfer_req()*.**

5.7.5.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

usbdi_intr_bulk_xfer_ack_op_t usbdi_intr_bulk_xfer_ack_unused;
```

5.7.5.2 Description

usbdi_intr_bulk_xfer_ack_unused() maybe used in the *usbdi_ldd_pipe_ops_t* as the *usbdi_intr_bulk_xfer_ack_op* if the LDD will never call *usbdi_intr_bulk_xfer_req()*.

5.7.6 usbdi_intr_bulk_xfer_nak_unused()

***usbdi_intr_bulk_xfer_nak_unused()* - Proxy for *usbdi_intr_bulk_xfer_req()*.**

5.7.6.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

usbdi_intr_bulk_xfer_nak_op_t usbdi_intr_bulk_xfer_nak_unused;
```

5.7.6.2 Description

usbdi_intr_bulk_xfer_nak_unused() maybe used in the *usbdi_ldd_pipe_ops_t* as the *usbdi_intr_bulk_xfer_nak_op* if the LDD will never call *usbdi_intr_bulk_xfer_req()*.

5.8 USB Control Transfer Requests

This section contains descriptions for the following control block, functions, and *typedefs*:

- *usbdi_control_xfer_cb_t*
- *usbdi_control_xfer_req()*
- *usbdi_control_xfer_ack_op_t*
- *usbdi_control_xfer_ack_unused()*

The following function and *typedef* are used only by the USBD (not by the LDD). Callee-side interfaces may be derived from the caller-side interfaces by removing the first parameter, and vice versa. They are not described in this specification but are included in "Appendix A: Include File (open_usbdi.h)" for completeness:

- *usbdi_control_xfer_req_op_t*
- *usbdi_control_xfer_ack()*

5.8.1 usbdi_control_xfer_cb_t

***usbdi_control_xfer_cb_t* – Used in transferring data to or from a control endpoint.**

5.8.1.1 Synopsis

```
typedef struct {

      udi_cb_t gcb;     /* UDI generic control block */

      /*
       * The “request” union is provided to support control endpoints
       * other than the default that may not use the usb_device_request_t
       * request format.  This is the data that will be sent to the
       * device during the set up phase of the control request.
       */
      union {
            /* Used by the default endpoint */
            usb_device_request_t device_request;

            /* Used by control endpoints other than the default */
            udi_ubit8_t request[8];
      } request;

      /*
       * data_buf points to the associated buffer to be used during the
       * data phase transfer (if any).  The LDD sets data_buf->buf_size
       * to the maximum number of bytes that may be transferred during the
       * data phase.  To specify a zero length data transfer
       * data_buf->buf_size shall be set by the LDD to zero.
       *
       * On completion the USBD sets data_buf->buf_size
       * with the actual number of bytes transferred.
       */
      udi_buf_t *data_buf;

      /*
       * Request timeout value in milliseconds.  Request timeout periods
       * begin when the USBD receives the control block
       * and NOT when the host controller issues the request to the
       * device.  A timeout value of zero specifies an infinite timeout
       * period.  If a request times out, it will be completed by the
       * USBD with a status of UDI_STAT_TIMEOUT.  Non-default pipes will be
       * left in the USBDI_STATE_STALLED state.  Default pipe’s state will
       * not be changed, remaining in the USBDI_STATE_ACTIVE state.
       */
      udi_ubit32_t timeout;

      /*
       * Flags specific to this control request.  The LDD shall set
       * the direction of the transfer during the data phase (if any).
       * by setting USBDI_XFER_IN or USBDI_XFER_OUT.
       * USBDI_XFER_SHORT_OK is also a valid flag.
       * See usbdi_intr_bulk_xfer_cb_t for flag defines.
```

```
         */
        udi_ubit8_t xfer_flags;

} usbdi_control_xfer_cb_t;

#define     USBDI_CONTROL_XFER_CB_NUM     3
```

5.8.1.2 Description

The control transfer control block is used in OpenUSBDI operations that involve transferring data to and from control endpoints.

In order to use this type of control block it must be associated with a control block index by including `USBDI_CONTROL_XFER_CB_NUM` in a `udi_cb_init_t` in the driver's `udi_init_info`.

5.8.2 usbdi_control_xfer_req()

***usbdi_control_xfer_req()* – Called by the LDD to initiate a transfer with a control endpoint.**

5.8.2.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

void usbdi_control_xfer_req(
            usbdi_control_xfer_cb_t *cb );
```

5.8.2.2 Arguments

usbdi_control_xfer_cb_t ****cb***	Pointer to control block that was allocated by *udi_cb_alloc()* given a cb_idx that was previously associated with USBDI_CONTROL_XFER_CB_NUM and given a LDD's pipe channel.

5.8.2.3 Description

usbdi_control_xfer_req() is called by the LDD to communicate with a control endpoint. This may be the default endpoint or any other control endpoint.

If the CB is being sent to the default endpoint, the request shall be set up using the *usb_device_request_t*. If the CB is being sent to a control endpoint other than the default, the LDD may use either the *usb_device_request_t* or the *request* array. Regardless, the number of bytes in the request shall total eight. (See the Universal Serial Bus Specification for more information on control device requests.)

The *data_buf->buf_size* field specifies the number of bytes to transfer. It may be set to zero ("0") if no data is to be transferred. It shall be set to a positive, non-zero value if data is to be transferred.

The *data_buf* field of the CB points to the buffer that contains or will contain the data. If *data_buf->buf_size* is greater than zero, *data_buf* shall point to a buffer that is large enough to accommodate *data_buf->buf_size* bytes.

The *timeout* field of the CB specifies the number of milliseconds that may pass from the time the CB is received by the USBD until the time the CB is passed to *usbdi_control_xfer_ack_op_t*. This value may be zero ("0") if an infinite timeout period is needed. In this case, the USBD will never time out the request.

The *xfer_flags* field of the CB shall be set to zero ("0"), USBDI_XFER_SHORT_OK, USBDI_XFER_IN, and/or USBDI_XFER_OUT. USBDI_XFER_IN specifies that, during the data phase of the control transfer, data will be transferred from the device to *data_buf*. USBDI_XFER_OUT specifies that, during the data phase, data will be transferred from *data_buf* to the device. If *data_buf->buf_size* is greater than zero, USBDI_XFER_IN or USBDI_XFER_OUT shall be set (only one may be set). If USBDI_XFER_SHORT_OK is set, and the device transfers less than *data_buf->buf_size* bytes, the control block will complete with a status of UDI_OK. See Section 5.8.3, *usbdi_control_xfer_ack_op_t,* for more information.

The USBD may queue the request on the pipe if the pipe's state is USBDI_STATE_ACTIVE or USBDI_STATE_STALLED. The request will be sent to the endpoint if the pipe's state is USBDI_STATE_ACTIVE. See Section 5.17.3, *usbdi_pipe_state_get_req(),* for more information.

5.8.3 usbdi_control_xfer_ack_op_t

***usbdi_control_xfer_ack_op_t* – *typedef* for LDD supplied function called by the USBD when a transfer to a control endpoint completes with or without an error.**

5.8.3.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

typedef
void usbdi_control_xfer_ack_op_t(
            usbdi_control_xfer_cb_t *cb,
            udi_status_t status );
```

5.8.3.2 Arguments

usbdi_control_xfer_cb_t ****cb***	Pointer to the control block that was passed to *usbdi_control_xfer_req()*.
udi_status_t ***status***	See table below.

RETURNED STATUS	**DESCRIPTION**
UDI_OK	The request completed without an error.
USBDI_STAT_STALL	The device responded with a USB stall.
UDI_STAT_ABORTED	The request was successfully aborted as a result of *udi_channel_op_abort()*, *usbdi_pipe_abort_req()*, or *usbdi_intfc_abort_req()*.
UDI_STAT_INVALID_STATE	The pipe is in the USBDI_STATE_IDLE state.
UDI_STAT_DATA_OVERRUN	The device attempted to transfer more data than requested.
UDI_STAT_DATA_UNDERRUN	The device transferred less data than requested and USBDI_XFER_SHORT_OK was not set.
UDI_STAT_TIMEOUT	The timeout period for the request expired.
UDI_STAT_NOT_RESPONDING	The device is not responding or is inaccessible.
UDI_STAT_DATA_ERROR	An error occurred during the data transfer. The error may have been due to a bit stuffing error, data toggle

	mismatch, unexpected packet ID, etc.
UDI_STAT_MISTAKEN_IDENTITY	The CB contains a field with invalid data.

5.8.3.3 Description

The LDD shall supply a *usbdi_control_xfer_ack_op_t* function in its *usbdi_ldd_pipe_ops_t*. This function is called in response to the LDD's calling *usbdi_control_xfer_req()* when the operation completes with or without an error.

If USBDI_XFER_SHORT_OK was set in the *xfer_flags* of the *usbdi_control_xfer_cb_t*, then *data_buf->buf_size* shall be examined to determine the actual number of bytes transferred. In all cases, *data_buf->buf_size* will contain the actual number of bytes transferred during the data phase.

If an error occurred the state of the pipe shall be set by the USBD to USBDI_STATE_STALLED before this function is called, unless the pipe was already in the USBDI_STATE_STALLED state when *usbdi_control_xfer_req()* was called. See Section 5.17, "Getting and Setting Pipe States", for more information.

The LDD shall either free the CB by calling *udi_cb_free()* or reuse the CB for another call to *usbdi_control_xfer_req()*.

5.8.4 usbdi_control_xfer_ack_unused()

***usbdi_control_xfer_ack_unused()* - Proxy for *usbdi_control_xfer_req()*.**

5.8.4.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

usbdi_control_xfer_ack_op_t usbdi_control_xfer_ack_unused;
```

5.8.4.2 Description

usbdi_control_xfer_ack_unused() may be used in the *usbdi_ldd_pipe_ops_t* as the *usbdi_control_xfer_ack_op* if the LDD will never call *usbdi_control_xfer_req()*.

5.9 USB Isochronous Transfer Requests

This section contains descriptions for the following functions and *typedefs*:

- *usbdi_isoc_frame_request_t*
- *usbdi_isoc_xfer_cb_t*
- *usbdi_isoc_xfer_req()*
- *usbdi_isoc_xfer_ack_op_t*
- *usbdi_isoc_xfer_nak_op_t*
- *usbdi_isoc_xfer_ack_unused()*
- *usbdi_isoc_xfer_nak_unused()*

The following functions and *typedef* are used only by the USBD (not by the LDD). Callee-side interfaces may be derived from the caller-side interfaces by removing the first parameter, and vice versa. They are not described in this specification but are included in "Appendix A: Include File (open_usbdi.h)" for completeness:

- *usbdi_isoc_xfer_req_op_t*
- *usbdi_isoc_xfer_ack()*
- *usbdi_isoc_xfer_nak()*

5.9.1 usbdi_isoc_frame_request_t

An array of structures of type *usbdi_isoc_frame_request_t* will be allocated automatically when the *usbdi_isoc_xfer_cb_t* is allocated. Each element of this array describes a single isochronous transfer to / from the logical USB device.

The *usbdi_isoc_frame_request_t* contains two fields, one describing the length of the data to be transferred (specified by the LDD), and one that will receive the status of the transfer (filled in by the USBD on completion of the transfer operation). Figure 11 shows this arrangement.

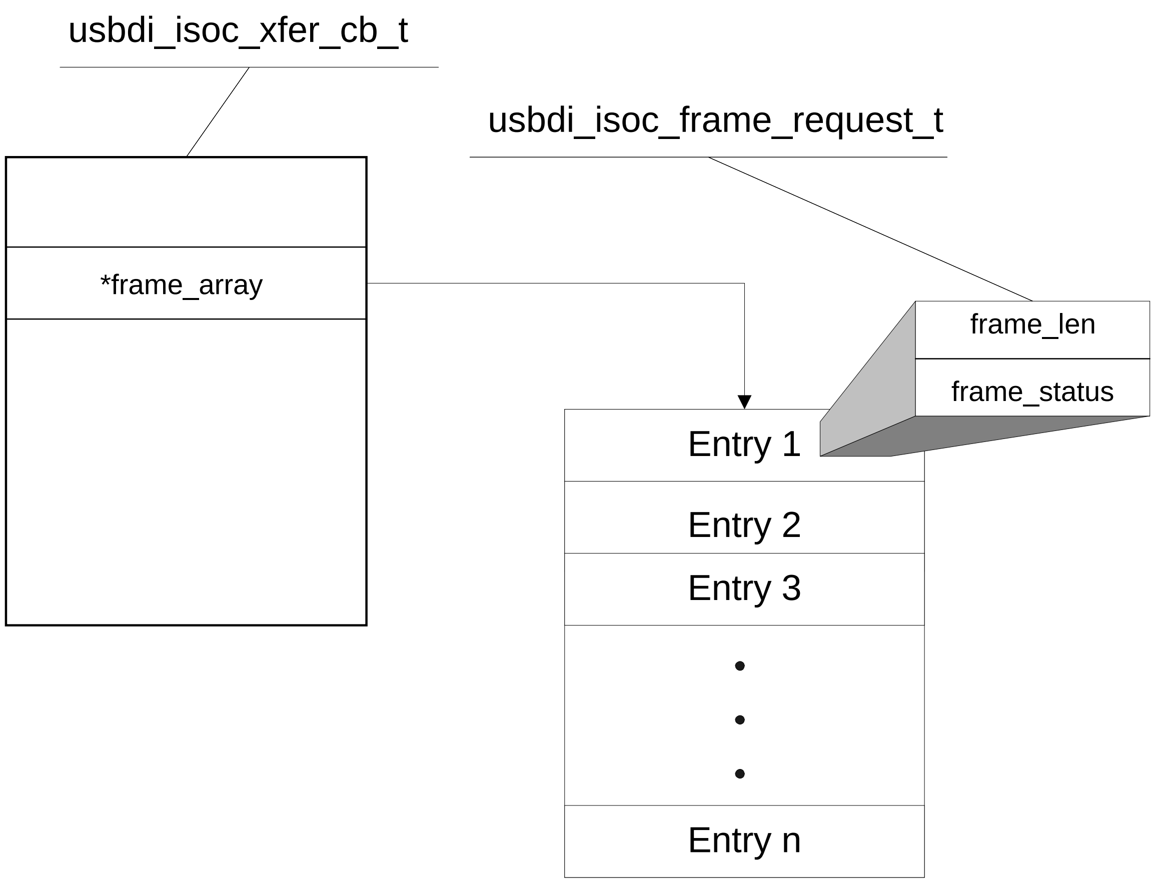

Figure 11

The number of elements allocated for the array is dependant on the *inline_size* field of the LDD's *udi_cb_init_t* structure for the isochronous control block. In this field, the LDD fills in the number of bytes that is to be allocated for the entire array :

```
<inline size> = <number of elements> *
                 sizeof(usbdi_isoc_frame_request_t)
```

```
typedef struct {
      udi_ubit32_t frame_len; /* Set by the LDD to the number of bytes to
                               * transfer in the frame.  Set by the USBD
                               * on completion with the actual number of
                               * bytes transferred in the frame.
                               */
      udi_status_t frame_status; /* Per frame status set by the USBD */
} usbdi_isoc_frame_request_t;
```

5.9.2 usbdi_isoc_xfer_cb_t

***usbdi_isoc_xfer_cb_t* – Used in transferring data to or from an isochronous endpoint.**

5.9.2.1 Synopsis

```
typedef struct {

      udi_cb_t gcb;     /* UDI generic control block */

      /*
       * data_buf shall be set by the LDD to a valid buffer where data
       * will be transferred to/from.  This buffer shall be large enough
       * to store all data that will be transferred as described by the
       * frame_array.
       */
      udi_buf_t *data_buf;

      /*
       * The frame_array will be allocated automatically when the
       * usbdi_isoc_xfer_cb_t is allocated.  The frame_len field for
       * each valid element in the array shall be set by the LDD.  It
       * is legal to specify a zero frame_len value.
       *
       * The size of this memory area is set by the LDD via the
       * inline_size member of the relevant udi_cb_init_t, and shall be
       * a multiple of the size of a single element of the array
       * (inline_size = #elements * sizeof(usbdi_isoc_frame_request_t))
       *
       * The LDD shall not modify or deallocate the memory pointed to
       * by frame_array
       */
      usbdi_isoc_frame_request_t *frame_array;

      /*
       * frame_count is the number of valid entries in the frame_array.
       * This field is set by the LDD and may not exceed the maximum number
       * of entries in the array (see usbdi_isoc_xfer_cb_init()).
       */
      udi_ubit8_t frame_count;

      /*
       * Flags specific to this request.  The USBDI_XFER_ASAP flag is
       * used to specify that the request may be started with the next
       * available frame.  If this flag is set, the frame_number field will
       * be ignored by the USBD.  The direction of the data is specified
       * with USBDI_XFER_IN or USBDI_XFER_OUT.
       * USBDI_XFER_SHORT_OK is also a valid flag.
       * See usbdi_intr_bulk_xfer_cb_t for flag defines
       */
      udi_ubit8_t xfer_flags;
#define USBDI_XFER_ASAP (1<<4)

      /*
```

```
       * frame_number is set by the LDD to the frame number that this
       * request may begin with (see usbdi_frame_number_get_req()).  It is
       * set by the USBD at request completion with the frame number at
       * completion time.  This frame number is provided to allow LDDs to
       * synchronize requests queued to different isochronous pipes.
       */
      udi_ubit32_t frame_number;
} usbdi_isoc_xfer_cb_t;
#define USBDI_ISOC_XFER_CB_NUM      4
```

Description

The isochronous transfer control block is used in OpenUSBDI operations that involve transferring data to and from isochronous endpoints.

In order to use this type of control block it must be associated with a control block index by including `USBDI_ISOC_XFER_CB_NUM` in a `udi_cb_init_t` in the driver's `udi_init_info`.

The size of the inline memory area pointed to by **frame_array** must be specified using the **inline_size** member of that *udi_cb_init_t* structure (see Chapter 10, “*Initialization*”, of the UDI Core Specification). This size must be a multiple of the size of a single array element (*usbdi_isoc_frame_request_t*), and is given in bytes.

The LDD shall not modify or deallocate the memory pointed to by **frame_array**.

5.9.3 usbdi_isoc_xfer_req()

***usbdi_isoc_xfer_req()* – Called by the LDD to initiate a transfer with an isochronous endpoint.**

5.9.3.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

void usbdi_isoc_xfer_req(
            usbdi_isoc_xfer_cb_t *cb );
```

5.9.3.2 Arguments

usbdi_isoc_xfer_cb_t ***cb***	Pointer to a control block that was allocated with *udi_cb_alloc()* given a cb_idx that was previously associated with USBDI_ISOC_XFER_CB_NUM and given a LDD's pipe channel.

5.9.3.3 Description

usbdi_isoc_xfer_req() is called by the LDD to transfer data with an isochronous endpoint. The pipe described by ***pipe_channel*** shall have been spawned by the LDD with a call to *udi_channel_spawn()*.

The *frame_array* of the CB points to an array of *usbdi_isoc_frame_request_t* elements. The number of valid elements in the array will be *frame_count*, each of which must be set up by the LDD. The *frame_len* field of each valid element shall be set (a *frame_len* of zero ("0") is valid). The *frame_status* will be set by the USBD.

The USBD shall determine the buffer offset for each frame element to be transferred *before* data is received from the device. If the transfer direction is from the logical device to the LDD, isochronous transfers with the USBDI_SHORT_XFER_OK flag set are subject to "holes", or regions of invalid data. For example, consider an LDD that requests *n* frames of 16 bytes of data, and only receives *n* frames of 8 bytes of data. Because the data buffers are set up before the data is received, each frame of data is deposited in the LDD's buffer at offsets that are multiples of 16 (buf + 0, buf + 16, buf + 32). LDD designers shall not assume that their buffers contain contiguous device data upon isochronous request completion.

The *data_buf* of the CB shall point to a buffer that can accommodate the sum of the *frame_len* fields from the *frame_array*. The *data_buf* shall be set to NULL by the LDD if the sum of the *frame_len* fields is zero ("0").

The *frame_count* shall be set with the number of valid frames set up by the LDD in *frame_array*.

The USBD allows LDDs to synchronize requests based on a reference frame number (see Section 5.9.5, "USB Isochronous Frame Number Determination", for more information). The LDD may specify the frame number when a given request may be issued to the isochronous endpoint by setting *frame_number* to a value using the value returned by *usbdi_frame_number_get_req()* as a base. The LDD may specify that the request shall be issued right away by setting USBDI_XFER_ASAP in the *xfer_flags* field. In this case, *frame_number* will not be examined by the USBD.

The *xfer_flags* field of the CB shall also be set with the direction of the data transfer (if data is to be transferred) USBDI_XFER_IN specifies that data will be transferred from the device to *data_buf*. USBDI_XFER_OUT specifies that data will be transferred from *data_buf* to the device.

The USBD may queue the control block on the pipe if the pipe's state is USBDI_STATE_ACTIVE or USBDI_STATE_STALLED. The USBD may issue the request to the endpoint if the state of the pipe is USBDI_STATE_ACTIVE. See Section 5.17.3, *usbdi_pipe_state_get_req(),* for more information.

5.9.4 usbdi_isoc_xfer_ack_op_t

***usbdi_isoc_xfer_ack_op_t* – *typedef* for LDD supplied function called by the USBD when an isochronous transfer operation completes without an error.**

5.9.4.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

typedef
void usbdi_isoc_xfer_ack_op_t(
            usbdi_isoc_xfer_cb_t *cb );
```

5.9.4.2 Arguments

usbdi_isoc_xfer_cb_t ****cb***	Pointer to control block that was passed to *usbdi_isoc_xfer_req()*.

5.9.4.3 Description

The LDD shall supply a *usbdi_isoc_xfer_ack_op_t* function in its *usbdi_ldd_pipe_ops_t*. This function is called in response to the LDD's calling *usbdi_isoc_xfer_req()* when the operation completes without an error.

The *frame_number* of CB will be set by the USBD with the reference frame number at the time *usbdi_isoc_xfer_ack_op_t* is called.

Each entry in *frame_array* of CB may be examined by the LDD to determine the number of bytes transferred during that frame and the status of the transfer.

5.9.5 usbdi_isoc_xfer_nak_op_t

***usbdi_isoc_xfer_nak_op_t – typedef* for LDD supplied function called by the USBD when an isochronous transfer operation completes with an error.**

5.9.5.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

typedef
void usbdi_isoc_xfer_nak_op_t(
            usbdi_isoc_xfer_cb_t *cb,
            udi_status_t status );
```

5.9.5.2 Arguments

usbdi_isoc_xfer_cb_t ***cb***	Pointer to control block that was passed to *usbdi_isoc_xfer_req()*.
udi_status_t ***status***	See table below.

RETURNED STATUS	DESCRIPTION
UDI_STAT_ABORTED	The request was successfully aborted as a result of *udi_channel_op_abort()*, *usbdi_pipe_abort_req()*, or *usbdi_intfc_abort_req()*.
UDI_STAT_HW_PROBLEM	One or more entries in frame_array have an error.
UDI_STAT_INVALID_STATE	The pipe is in the USBDI_STATE_IDLE state.
UDI_STAT_MISTAKEN_IDENTITY	A field in the CB contains invalid data.

5.9.5.3 Description

The LDD shall supply a *usbdi_isoc_xfer_nak_op_t* function in its *usbdi_ldd_pipe_ops_t*. This function is called in response to the LDD's calling *usbdi_isoc_xfer_req()* when the operation completes with an error.

The state of the pipe will be set to USBDI_STATE_STALLED by the USBD before this function is called. See Section 5.17, "Getting and Setting Pipe States", for more information.

The *frame_number* will be set by the USBD with the reference frame number at the time *usbdi_isoc_xfer_ack_op_t* is called.

Each entry in *frame_array* may be examined by the LDD to determine the number of bytes transferred during that frame and the status of the transfer.

The *frame_status* of each *usbdi_isoc_frame_request_t* may contain the following:

FRAME STATUS	**DESCRIPTION**
UDI_OK	The frame transfer completed properly.
USBDI_STAT_STALL	The device responded with a USB stall.
UDI_STAT_DATA_OVERRUN	The device attempted to transfer more data than requested.
UDI_STAT_DATA_UNDERRUN	The device transferred less data than requested and USBDI_XFER_SHORT_OK was not set in the *xfer_flags* field of the CB.
UDI_STAT_NOT_RESPONDING	The device is not responding or is inaccessible.
UDI_STAT_DATA_ERROR	An error occurred during the data transfer. The error may have been due to a bit stuffing error, data toggle mismatch, unexpected packet ID, etc.

The LDD shall either free the CB by calling *udi_cb_free()* or reuse it for another call to *usbdi_isoc_xfer_req()*.

5.9.6 usbdi_isoc_xfer_ack_unused()

***usbdi_isoc_xfer_ack_unused()* - Proxy for *usbdi_isoc_xfer_req()*.**

5.9.6.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

usbdi_isoc_xfer_ack_op_t usbdi_isoc_xfer_ack_unused;
```

5.9.6.2 Description

usbdi_isoc_xfer_ack_unused() may be used in the *usbdi_ldd_pipe_ops_t* as the *usbdi_isoc_xfer_ack_op* if the LDD will never call *usbdi_isoc_xfer_req()*.

5.9.7 usbdi_isoc_xfer_nak_unused()

***usbdi_isoc_xfer_nak_unused()* - Proxy for *usbdi_isoc_xfer_req()*.**

5.9.7.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

usbdi_isoc_xfer_nak_op_t usbdi_isoc_xfer_nak_unused;
```

5.9.7.2 Description

usbdi_isoc_xfer_nak_unused() may be used in the *usbdi_ldd_pipe_ops_t* as the *usbdi_isoc_xfer_nak_op* if the LDD will never call *usbdi_isoc_xfer_req()*.

5.10 USB Isochronous Frame Number Determination

The following section describes *typedefs* and functions that provide the isochronous LDD the ability to synchronize control blocks based on a time-based reference frame number. This frame number may not be the actual frame number from the USB host controller. It is a pseudo-frame number provided by the USBD that will allow isochronous LDDs to synchronize pipes that may be on the same USB or on different USBs.

This section contains descriptions for the following functions and *typedef*:

- *usbdi_frame_number_req()*
- *usbdi_frame_number_ack_op_t*
- *usbdi_frame_number_ack_unused()*

The following function and *typedef* are used only by the USBD (not by the LDD) and are not described in this specification. Callee-side interfaces may be derived from the caller-side interfaces by removing the first parameter, and vice versa. They are included in "Appendix A: Include File (open_usbdi.h)" for completeness:

- *usbdi_frame_number_req_op_t*
- *usbdi_frame_number_ack()*

5.10.1 usbdi_frame_number_req()

***usbdi_frame_number_req()* – Called by the LDD to initiate a request to the USBD to determine the current reference frame number.**

5.10.1.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

void usbdi_frame_number_req(
            usbdi_misc_cb_t *cb );
```

5.10.1.2 Arguments

usbdi_misc_cb_t ***cb***	Pointer to a control block that was allocated with *udi_cb_alloc()* given a cb_idx that was previously associated with USBDI_MISC_CB_NUM and given a LDD's interface channel.

5.10.1.3 Description

usbdi_frame_number_req() is called by the LDD to determine the current reference frame number. This frame number might not be based on the actual USB host controller frame number. It provides isochronous LDDs with the ability to synchronize control blocks that are sent to different pipes. This operation is performed on an interface channel.

This request does not generate any activity on the USB.

5.10.2 usbdi_frame_number_ack_op_t

***usbdi_frame_number_ack_op_t – typedef* for LDD supplied function called by the USBD on completion of a request for the current reference frame number.**

5.10.2.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

typedef
void usbdi_frame_number_ack_op_t(
             usbdi_misc_cb_t *cb,
             udi_ubit32_t frame_number );
```

5.10.2.2 Arguments

usbdi_misc_cb_t ***cb***	Pointer to a control block that was passed to *usbdi_frame_number_req()*.
udi_ubit32_t ***frame_number***	Reference frame number returned by the USBD.

5.10.2.3 Description

The LDD shall supply a *usbdi_frame_number_ack_op_t* function in its *usbdi_ldd_intfc_ops_t*. This function is called in response to the LDD's calling *usbdi_frame_number_req()*. The result is ***frame_number*** being set by the USBD with the current reference frame number.

The LDD shall either free the CB by calling *udi_cb_free()* or reuse the CB.

5.10.3 usbdi_frame_number_ack_unused()

***usbdi_frame_number_ack_unused()* - Proxy for *usbdi_frame_number_req()*.**

5.10.3.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

usbdi_frame_number_ack_op_t usbdi_frame_number_ack_unused;
```

5.10.3.2 Description

usbdi_frame_number_ack_unused() may be used in the *usbdi_ldd_intfc_ops_t* as the *usbdi_frame_number_ack_op* if the LDD will never call *usbdi_frame_number_req()*.

5.11 USB Isochronous Control Block Initialization

The following section describes a utility function used to set up isochronous endpoint control blocks.

This section contains a description for the following function:

- *usbdi_frame_distrib()*

5.11.1 usbdi_isoc_frame_distrib

5.11.1.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

udi_status_t usbdi_isoc_frame_distrib(
                                     usbdi_isoc_xfer_cb_t *cb );
```

5.11.1.2 Arguments

usbdi_isoc_xfer_cb_t ****cb***	Pointer to a control block that was allocated with *udi_cb_alloc()* given a cb_idx that was previously associated with USBDI_ISOC_XFER_CB_NUM and given the LDD's bind channel

RETURNED STATUS	**DESCRIPTION**
UDI_OK	The request was successful.
UDI_STAT_NOT_UNDERSTOOD	One or more control block fields required by this routine were not valid. This means either: 1. The `data_buf` field does not contain a valid `udi_buf_t`, or 2. The `frame_count` field did not contain a valid value.

5.11.1.3 Description

usbdi_isoc_frame_distrib() is an optional utility routine that may be used by LDDs to prepare the array of frame buffers in an isochronous control block. This routine is by no means required, provided the LDD sets up the frame buffer array before attempting to transfer data over the isochronous USB pipe.

usbdi_isoc_frame_distrib() is called by the LDD to calculate the frame length size of each frame that will be involved in an isochronous transfer. This may help isolate the LDD from the details of the `usbdi_isoc_frame_request_t`.

This routine requires two fields in the `usbdi_isoc_xfer_cb_t` field to be filled in by the LDD before calling this routine:

- `data_buf`
  Must contain the `udi_buf_t` for the data to be transferred via the isochronous pipe.
- `frame_count`
  Must contain the number of frames over which the data is to be divided.

This routine will use this information to set the `frame_len` of each `usbdi_isoc_frame_request_t` within the array referenced by `frame_array`. This routine shall set the length of each field as follows:

```
for each frame except the last:
   cb->frame[n].frame_len = cb->data_buf.buf_size / cb->frame_count

for the last frame:
   cb->frame[last].frame_len = cb->data_buf.buf_size % cb->frame_count
```

LDDs requireing more precise frame control will need to do their own frame buffer manipulation.

This utility routine does not generate any activity on the USB, and returns synchronously.

5.12 USB Device Speed Determination

The following section describes the *typedef* and functions that provide the LDD the ability to determine the speed of its associated device.

This section contains descriptions for the following functions and *typedef*:

- *usbdi_device_speed_req()*
- *usbdi_device_speed_ack_op_t*
- *usbdi_device_speed_ack_unused()*

The following function and *typedef* are used only by the USBD (not by the LDD) and are not described in this specification. Callee-side interfaces may be derived from the caller-side interfaces by removing the first parameter, and vice versa. They are included in “Appendix A: Include File (open_usbdi.h)” for completeness:

- *usbdi_device_speed_req_op_t*
- *usbdi_device_speed_ack()*

5.12.1 usbdi_device_speed_req()

***usbdi_device_speed_req()* – Called by the LDD to initiate a request to the USBD to determine the speed of the device.**

5.12.1.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

void usbdi_device_speed_req(
            usbdi_misc_cb_t *cb );
```

5.12.1.2 Arguments

usbdi_misc_cb_t ***cb***	Pointer to a control block that was allocated with *udi_cb_alloc()* given a cb_idx that was previously associated with USBDI_MISC_CB_NUM and given the LDD's bind channel.

5.12.1.3 Description

usbdi_device_speed_req() is called by the LDD to determine the speed of the USB device associated with ***intfc_channel***. This operation is performed on the bind channel.

This request does not generate any activity on the USB.

5.12.2 usbdi_device_speed_ack_op_t

***usbdi_device_speed_ack_op_t – typedef* for LDD supplied function called by the USBD on completion of a request for the device speed.**

5.12.2.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

typedef
void usbdi_device_speed_ack_op_t(
            usbdi_misc_cb_t *cb,
            udi_ubit8_t device_speed );
```

5.12.2.2 Arguments

usbdi_misc_cb_t ****cb***	Pointer to a control block that was passed to *usbdi_device_speed_req()*.
udi_ubit8_t ***device_speed***	One of the following values defined in the include file open_usbdi.h: USBDI_DEVICE_SPEED_LOW, USBDI_DEVICE_SPEED_FULL, USBDI_DEVICE_SPEED_HIGH.

5.12.2.3 Description

The LDD shall supply a *usbdi_device_speed_ack_op_t* function in its *usbdi_ldd_intfc_ops_t*. This function is called in response to the LDD's calling *usbdi_device_speed_req()*. The result is ***device_speed*** being set by the USBD with the speed of the device (USBDI_DEVICE_SPEED_LOW, USBDI_DEVICE_SPEED_FULL, or USBDI_DEVICE_SPEED_HIGH).

The LDD shall either free the CB by calling *udi_cb_free()* or reuse the CB.

5.12.3 usbdi_device_speed_ack_unused()

***usbdi_device_speed_ack_unused()* – Proxy for *usbdi_device_speed_req()*.**

5.12.3.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

usbdi_device_speed_ack_op_t usbdi_device_speed_ack_unused;
```

5.12.3.2 Description

usbdi_device_speed_ack_unused() may be used in the *usbdi_ldd_intfc_ops_t* as the *usbdi_device_speed_ack_op* if the LDD will never call *usbdi_device_speed_req()*.

5.13 Resetting a Device

For specific information on LDD (or any driver) initialization within UDI refer to the “Management Metalanguage” section of the UDI Core Specification. Note that performing a reset of a USB device will cause the device to be re-enumerated, resetting the device to its default configuration. This could cause the current LDDs for the device to be unbound.

This section contains descriptions for the following functions and *typedef:*

- *usbdi_reset_device_req()*
- *usbdi_reset_device_ack_op_t*
- *usbdi_reset_device_ack_unused()*

The following function and *typedef* are used only by the USBD (not by the LDD) and are not described in this specification. Callee-side interfaces may be derived from the caller-side interfaces by removing the first parameter, and vice versa. They are included in “Appendix A: Include File (open_usbdi.h)” for completeness:

- *usbdi_reset_device_req_op_t*
- *usbdi_reset_device_ack()*

5.13.1 usbdi_reset_device_req()

***usbdi_reset_device_req()* – Called by the LDD to initiate a reset of a USB device.**

5.13.1.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

void usbdi_reset_device_req(
            usbdi_misc_cb_t *cb );
```

5.13.1.2 Arguments

usbdi_misc_cb_t ***cb***	Pointer to control block that was allocated with *udi_cb_alloc()* given a cb_idx that was previously associated with USBDI_MISC_CB_NUM and given a LDD's interface channel.

5.13.1.3 Description

usbdi_reset_device_req() is called by the LDD to reset the USB port to which the device controlling the interface associated with ***intfc_channel*** is attached. This function may only be used in the most extreme error recovery paths. The result of the device being reset is that the device will be set to its default configuration and the LDD(s) will be unbound (resulting in all channels being closed) and rebound. Refer to the Universal Serial Bus Specification for more information on USB device reset. This operation is performed on an interface channel.

5.13.2 usbdi_reset_device_ack_op_t

***usbdi_reset_device_ack_op_t* – *typedef* for LDD supplied function called by the USBD when a USB device reset operation completes.**

5.13.2.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

typedef
void usbdi_reset_device_ack_op_t(
            usbdi_misc_cb_t *cb );
```

5.13.2.2 Arguments

usbdi_misc_cb_t ***cb***	Pointer to a control block that was passed to *usbdi_reset_device_req()*.

5.13.2.3 Description

The LDD shall supply a *usbdi_reset_device_ack_op_t* function in its *usbdi_ldd_intfc_ops_t*. This function is called in response to the LDD's calling *usbdi_reset_device_req()*.

The LDD shall either free the CB by calling *udi_cb_free()* or reuse the CB.

5.13.3 usbdi_reset_device_ack_unused()

***usbdi_reset_device_ack_unused()* - Used by the LDD to specify to the USBD that it will never perform a *usbdi_reset_device_req()*.**

5.13.3.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

usbdi_reset_device_ack_op_t usbdi_reset_device_ack_unused;
```

5.13.3.2 Description

usbdi_reset_device_ack_unused() maybe used in the *usbdi_ldd_intfc_ops_t* as the *usbdi_reset_device_ack_op* if the LDD will never call *usbdi_reset_device_req()*.

5.14 Aborting a Pipe

During some error recovery situations an LDD may need to abort all transfer control blocks (*usbdi_intr_bulk_xfer_cb_t*, *usbdi_control_xfer_cb_t*, or *usbdi_isoc_xfer_cb_t*) that have been queued to a pipe.

The LDD initiates the pipe abort by calling *usbdi_pipe_abort_req()*. All of the requests outstanding on the pipe are returned. Then, the LDD's *usbdi_pipe_abort_ack_op_t* operation is called. On the completion of the pipe abort the pipe's state will be USBDI_STATE_STALLED. See Section 5.17, "Getting and Setting Pipe States", for more information.

This section contains descriptions for the following functions and *typedef*:

- *usbdi_pipe_abort_req()*
- *usbdi_pipe_abort_ack_op_t*
- *usbdi_pipe_abort_ack_unused()*

The following function and *typedef* are used only by the USBD (not by the LDD) and are not described in this specification. Callee-side interfaces may be derived from the caller-side interfaces by removing the first parameter, and vice versa. They are included in "Appendix A: Include File (open_usbdi.h)" for completeness:

- *usbdi_pipe_abort_req_op_t*
- *usbdi_pipe_abort_ack()*

5.14.1 usbdi_pipe_abort_req()

***usbdi_pipe_abort_req()* – Called by the LDD to initiate an abort of a pipe.**

5.14.1.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

void usbdi_pipe_abort_req(
            usbdi_misc_cb_t *cb );
```

5.14.1.2 Arguments

usbdi_misc_cb_t ***cb***	Pointer to a control block that was allocated with *udi_cb_alloc()* given a cb_idx that was previously associated with USBDI_MISC_CB_NUM and given a LDD's pipe channel.

5.14.1.3 Description

usbdi_pipe_abort_req() is called by the LDD to abort all transfer control blocks that have been queued to the pipe associated with ***pipe_channel***. All transfer control blocks that are successfully aborted will be returned to the appropriate LDD *xfer_ack_op_t* (for control endpoints) or *xfer_nak_op_t* function with a status of UDI_STAT_ABORTED. The pipe may be in any state when *usbdi_pipe_abort_req()* is called. This operation is performed on a pipe channel.

5.14.2 usbdi_pipe_abort_ack_op_t

***usbdi_pipe_abort_ack_op_t – typedef* for LDD supplied function called by the USBD when an abort pipe operation completes.**

5.14.2.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

typedef
void usbdi_pipe_abort_ack_op_t(
            usbdi_misc_cb_t *cb );
```

5.14.2.2 Arguments

usbdi_misc_cb_t ****cb***	Pointer to a control block that was passed to *usbdi_pipe_abort_req()*.

5.14.2.3 Description

The LDD shall supply a *usbdi_pipe_abort_ack_op_t* function in its *usbdi_ldd_pipe_ops_t*. This function is called in response to the LDD's calling *usbdi_pipe_abort_req()*. All transfer control blocks queued to the pipe will be returned to their appropriate LDD *xfer_ack_op_t* (for control endpoints) or *xfer_nak_op_t* function with a status of UDI_STAT_ABORTED before the LDD's *usbdi_pipe_abort_ack_op_t* operation is called. The *usbdi_misc_cb_t* is then returned to the LDD's *usbdi_pipe_abort_ack_op_t* function. On completion of the pipe abort, the state of the pipe will be USBDI_STATE_STALLED if the state of the pipe before the abort was either USBDI_STATE_ACTIVE or USBDI_STATE_STALLED. The state of the pipe after the abort will be USBDI_STATE_IDLE if the pipe was in the USBDI_STATE_IDLE state before the abort.

The LDD shall either free the CB by calling *udi_cb_free()* or reuse the CB.

5.14.3 usbdi_pipe_abort_ack_unused()

usbdi_pipe_abort_ack_unused() **- Proxy for** ***usbdi_pipe_abort_req().***

5.14.3.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

usbdi_pipe_abort_ack_op_t usbdi_pipe_abort_ack_unused;
```

5.14.3.2 Description

usbdi_pipe_abort_ack_unused() may be used in the *usbdi_ldd_pipe_ops_t* as the *usbdi_pipe_abort_ack_op* if the LDD will never call *usbdi_pipe_abort_req().*

5.15 Aborting an Interface

During some error recovery situations an LDD may need to abort all transfer control blocks (*usbdi_intr_bulk_xfer_cb_t*, *usbdi_control_xfer_cb_t*, or *usbdi_isoc_xfer_cb_t*) that have been queued to all pipes in an interface.

The LDD initiates the interface abort by calling *usbdi_intfc_abort_req()*. All control blocks will be aborted before the LDD's *usbdi_intfc_abort_ack_op_t* is called.

This section contains descriptions for the following functions and *typedef*:

- *usbdi_intfc_abort_req()*
- *usbdi_intfc_abort_ack_op_t*
- *usbdi_intfc_abort_ack_unused()*

The following function and *typedef* are used only by the USBD (not by the LDD) and are not described in this specification. Callee-side interfaces may be derived from the caller-side interfaces by removing the first parameter, and vice versa. They are included in "Appendix A: Include File (open_usbdi.h)" for completeness:

- *usbdi_intfc_abort_req_op_t*
- *usbdi_intfc_abort_ack()*

5.15.1 usbdi_intfc_abort_req()

***usbdi_intfc_abort_req()* – Called by the LDD to abort a USB interface.**

5.15.1.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

void usbdi_intfc_abort_req(
            usbdi_misc_cb_t *cb );
```

5.15.1.2 Arguments

usbdi_misc_cb_t ***cb***	Pointer to a control block that was allocated with *udi_cb_alloc()* given a cb_idx that was previously associated with USBDI_MISC_CB_NUM and given a LDD's interface channel.

5.15.1.3 Description

usbdi_intfc_abort_req() is called by the LDD to abort all transfer control blocks *(usbdi_intr_bulk_xfer_cb_t*, *usbdi_control_xfer_cb_t*, and / or *usbdi_isoc_xfer_cb_t*) that have been queued to all pipes controlled by the interface specified by ***intfc_channel***. The interface may be in any state when *usbdi_intfc_abort_req()* is called.

5.15.2 usbdi_intfc_abort_ack_op_t

***usbdi_intfc_abort_ack_op_t – typedef* for LDD supplied function called by the USBD on completion of an interface abort operation.**

5.15.2.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

typedef
void usbdi_intfc_abort_ack_op_t(
            usbdi_misc_cb_t *cb );
```

5.15.2.2 Arguments

usbdi_misc_cb_t ****cb***	Pointer to a control block that was passed to *usbdi_intfc_abort_req()*.

5.15.2.3 Description

The LDD shall supply a *usbdi_intfc_abort_ack_op_t* function in its *usbdi_ldd_intfc_ops_t*. This function is called in response to the LDD's calling *usbdi_intfc_abort_req()*. All transfer control blocks queued to all pipes controlled by the interface will be returned to the appropriate LDD *xfer_ack_op_t* (in the case of control pipe transfers) or *xfer_nak_op_t* (all other transfers) function with a status of UDI_STAT_ABORTED.

Only after all transfer control blocks have been returned will the *usbdi_misc_cb_t* be returned to the LDD's *usbdi_intfc_abort_ack_op_t* function. Once the interface abort has been completed (*usbdi_intfc_abort_ack_op_t* has been called) the interface will be in the USBDI_STATE_STALLED state if the interface state before the abort was USBDI_STATE_ACTIVE or USBDI_STATE_STALLED. The interface state after the abort will be USBDI_STATE_IDLE if it was USBDI_STATE_IDLE before the abort. See Section 5.18.5, "Getting and Setting Interface States", for more information.

The LDD shall either free the CB by calling *udi_cb_free()* or reuse the CB.

5.15.3 usbdi_intfc_abort_ack_unused()

***usbdi_intfc_abort_ack_unused()* - Proxy for *usbdi_intfc_abort_req()*.**

5.15.3.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

usbdi_intfc_abort_ack_op_t usbdi_intfc_abort_ack_unused;
```

5.15.3.2 Description

usbdi_intfc_abort_ack_unused() may be used in the *usbdi_ldd_intfc_ops_t* as the *usbdi_intfc_abort_ack_op* if the LDD will never call *usbdi_intfc_abort_req()*.

5.16 Getting and Setting States

Devices, interfaces, pipes, and endpoints all have a state associated with them. The following control block is used for setting and getting these states.

5.16.1 usbdi_state_cb_t

```
typedef struct {

      udi_cb_t gcb;            /* UDI generic control block */
      udi_ubit8_t state;       /* Current state or state being
                                * set
                                */
/*
 * usbdi_pipe_state_set_req(), usbdi_pipe_state_get_req(),
 * usbdi_intfc_state_set_req(), and usbdi_intfc_state_get_req() use
 * USBDI_STATE_ACTIVE, USBDI_STATE_STALLED, and USBDI_STATE_IDLE.
 *
 * usbdi_edpt_state_set_req() and usbdi_edpt_state_get_req() use
 * USBDI_STATE_ACTIVE and USBDI_STATE_HALTED.
 */
#define USBDI_STATE_ACTIVE          1
#define USBDI_STATE_STALLED         2
#define USBDI_STATE_IDLE            3
#define USBDI_STATE_HALTED          4

/*
 * usbdi_device_state_get_req() uses USBDI_STATE_CONFIGURED and
 * USBDI_STATE_SUSPENDED.  Note that a device may be configured and
 * suspended at the same time.
 */
#define USBDI_STATE_CONFIGURED      (1 << 1)
#define USBDI_STATE_SUSPENDED       (1 << 2)

} usbdi_state_cb_t;

#define USBDI_STATE_CB_NUM          4
```

5.17 Getting and Setting Pipe States

Pipes support three states:

USBDI_STATE_ACTIVE	The pipe will accept transfer requests and send requests to the endpoint.
USBDI_STATE_STALLED	The pipe will accept transfer requests but not send requests to the endpoint.
USBDI_STATE_IDLE	The pipe will neither accept transfer requests nor send requests to the endpoint.

This section contains descriptions for the following functions and *typedefs*:

- *usbdi_pipe_state_set_req()*
- *usbdi_pipe_state_set_ack_op_t*
- *usbdi_pipe_state_get_req()*
- *usbdi_pipe_state_get_ack_op_t*
- *usbdi_pipe_state_get_ack_unused()*

The following functions and *typedefs* are used only by the USBD (not by the LDD) and are not described in this specification. Callee-side interfaces may be derived from the caller-side interfaces by removing the first parameter, and vice versa. They are included in “Appendix A: Include File (open_usbdi.h)” for completeness:

- *usbdi_pipe_state_get_req_op_t*
- *usbdi_pipe_state_get_ack()*
- *usbdi_pipe_state_set_req_op_t*
- *usbdi_pipe_state_set_ack()*

5.17.1 usbdi_pipe_state_set_req()

***usbdi_pipe_state_set_req()* – Called by the LDD to set the state of a pipe.**

5.17.1.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

void usbdi_pipe_state_set_req(
            usbdi_state_cb_t *cb );
```

5.17.1.2 Arguments

usbdi_state_cb_t ***cb***	Pointer to a control block that was allocated with *udi_cb_alloc()* given a cb_idx that was previously associated with USBDI_STATE_CB_NUM and given a LDD's pipe channel.

5.17.1.3 Description

usbdi_pipe_state_set_req() is called by the LDD to change the state of the pipe channel associated with the control block. The new pipe state is set by the LDD in the *state* field of the CB. Valid values are USBDI_STATE_ACTIVE, USBDI_STATE_STALLED, and USBDI_STATE_IDLE.

The LDD shall not change the state of the default endpoint pipe. A *usbdi_pipe_state_set_req()* operation performed on the default ***pipe_channel*** is considered a no-op.

usbdi_pipe_state_set_req() shall be used only if the state of the interface is USBDI_STATE_ACTIVE. See Section 5.18.5, "Getting and Setting Interface States", for more information.

This operation is performed on a pipe channel.

5.17.2 usbdi_pipe_state_set_ack_op_t

***usbdi_pipe_state_set_ack_op_t – typedef* for LDD supplied function called by the USBD on the completion of a pipe state set operation.**

5.17.2.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

typedef
void usbdi_pipe_state_set_ack_op_t(
            usbdi_state_cb_t *cb,
            udi_status_t status );
```

5.17.2.2 Arguments

usbdi_state_cb_t ****cb***	Pointer to a control block that was passed to *usbdi_pipe_state_set_req()*.
udi_status_t ***status***	See table below.

RETURNED STATUS	**DESCRIPTION**
UDI_OK	The pipe state was successfully set.
UDI_STAT_MISTAKEN_IDENTITY	The *state* field of CB contained invalid data.
UDI_STAT_INVALID_STATE	The pipe state was not set, due to the interface state not being USBDI_STATE_ACTIVE.

5.17.2.3 Description

The LDD shall supply a *usbdi_pipe_state_set_ack_op_t* function in its *usbdi_ldd_pipe_ops_t*. This function is called in response to the LDD's calling *usbdi_pipe_state_set_req()*. The state of the pipe has been changed if ***status*** is UDI_OK.

The LDD shall either free the CB by calling *udi_cb_free()* or reuse the CB.

5.17.3 usbdi_pipe_state_get_req()

***usbdi_pipe_state_get_req()* – Called by the LDD to retrieve the state of a pipe.**

5.17.3.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

void usbdi_pipe_state_get_req(
            usbdi_state_cb_t *cb);
```

5.17.3.2 Arguments

usbdi_state_cb_t ***cb***	Pointer to a control block that was allocated with *udi_cb_alloc()* given a cb_idx that was previously associated with USBDI_STATE_CB_NUM and given a LDD's pipe channel.

5.17.3.3 Description

usbdi_pipe_state_get_req() is called by the LDD to get the state of the pipe channel associated with the control block.

usbdi_pipe_state_get_req() may be used at any time, regardless of the state of the interface.

This operation is performed on a pipe channel.

5.17.4 usbdi_pipe_state_get_ack_op_t

***usbdi_pipe_state_get_ack_op_t – typedef* for LDD supplied function called by the USBD on completion of a get pipe state operation.**

5.17.4.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

typedef
void usbdi_pipe_state_get_ack_op_t(
            usbdi_state_cb_t *cb );
```

5.17.4.2 Arguments

usbdi_state_cb_t ***cb***	Pointer to a control block that was passed to *usbdi_pipe_state_get_req()*.

5.17.4.3 Description

The LDD shall supply a *usbdi_pipe_state_get_ack_op_t* function in its *usbdi_ldd_pipe_ops_t*. This function is called in response to the LDD's calling *usbdi_pipe_state_get_req()*. The *state* returned in CB is one of the following values: USBDI_STATE_ACTIVE, USBDI_STATE_STALLED, or USBDI_STATE_IDLE.

The LDD shall either free the CB by calling *udi_cb_free()* or may reuse it.

5.17.5 usbdi_pipe_state_get_ack_unused()

***usbdi_pipe_state_get_ack_unused()* – Proxy for *usbdi_pipe_state_get_req()*.**

5.17.5.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

usbdi_pipe_state_get_ack_op_t usbdi_pipe_state_get_ack_unused;
```

5.17.5.2 Description

usbdi_pipe_state_get_ack_unused() may be used in the *usbdi_ldd_pipe_ops_t* as the *usbdi_pipe_state_get_ack_op* if the LDD will never call *usbdi_pipe_state_get_req()*.

5.18 Getting and Setting Endpoint States

Endpoints support two states:

USBDI_STATE_ACTIVE	The endpoint will accept transfer requests and send requests to the device.
USBDI_STATE_HALTED	The endpoint has been halted. See the Universal Serial Bus Specification for a description of endpoint halts and possible causes. This state is read-only.

This section contains descriptions for the following functions and *typedefs*:

- *usbdi_edpt_state_set_req()*
- *usbdi_edpt_state_set_ack_op_t*
- *usbdi_edpt_state_get_req()*
- *usbdi_edpt_state_get_ack_op_t*
- *usbdi_edpt_state_get_ack_unused()*

The following functions and *typedefs* are used only by the USBD (not by the LDD) and are not described in this specification. Callee-side interfaces may be derived from the caller-side interfaces by removing the first parameter, and vice versa. They are included in “Appendix A: Include File (open_usbdi.h)” for completeness:

- *usbdi_edpt_state_set_req_op_t*
- *usbdi_edpt_state_set_ack()*
- *usbdi_edpt_state_get_req_op_t*
- *usbdi_edpt_state_get_ack()*

5.18.1 usbdi_edpt_state_set_req()

***usbdi_edpt_state_set_req()* – Called by the LDD to set the state of an endpoint.**

5.18.1.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

void usbdi_edpt_state_set_req(
            usbdi_state_cb_t *cb );
```

5.18.1.2 Arguments

usbdi_state_cb_t ***cb***	Pointer to a control block that was allocated with *udi_cb_alloc()* given a cb_idx that was previously associated with USBDI_STATE_CB_NUM and given a LDD's pipe channel.

5.18.1.3 Description

usbdi_edpt_state_set_req() is called by the LDD to change the state of the USB endpoint associated with the pipe channel associated with the control block. The new endpoint state is set by the LDD in the *state* field of the CB. The only valid value is USBDI_STATE_ACTIVE.

In response to a *usbdi_edpt_state_set_req()*, the USBD issues a USB ClearFeature(ENDPOINT_HALT)[10] for the endpoint associated with the pipe channel and resets the data-toggle for the endpoint.

The LDD shall not clear an endpoint halt condition by sending a ClearFeature(ENDPOINT_HALT) device request to the default endpoint directly.

usbdi_edpt_state_set_req() may be used at any time, regardless of the state of the pipe or interface. Data loss may occur if the endpoint is active (not halted) when an *usbdi_edpt_state_set_req()* is performed.

This operation is performed on a pipe channel.

[10] See the Universal Serial Bus Specification for the meaning of this term.

5.18.2 usbdi_edpt_state_set_ack_op_t

***usbdi_edpt_state_set_ack_op_t* – *typedef* for LDD supplied function called by the USBD on completion of a set endpoint state operation.**

5.18.2.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

typedef
void usbdi_edpt_state_set_ack_op_t(
            usbdi_state_cb_t *cb,
            udi_status_t status );
```

5.18.2.2 Arguments

usbdi_state_cb_t ***cb***	Pointer to a control block that was passed to *usbdi_edpt_state_set_req()*.
udi_status_t ***status***	See table below.

RETURNED STATUS	DESCRIPTION
UDI_OK	The endpoint state was successfully set.
UDI_STAT_NOT_RESPONDING	The device is not responding or is inaccessible.
UDI_STAT_MISTAKEN_IDENTITY	The endpoint state was not set due to an invalid value in the CB.

5.18.2.3 Description

The LDD shall supply a *usbdi_edpt_state_set_ack_op_t* function in its *usbdi_ldd_pipe_ops_t*. This function is called in response to the LDD’s calling *usbdi_edpt_state_set_req()*.

The LDD shall either free the CB by calling *udi_cb_free()* or reuse the CB.

5.18.3 usbdi_edpt_state_get_req()

***usbdi_edpt_state_get_req()* – Called by the LDD to retrieve the state of an endpoint.**

5.18.3.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

void usbdi_edpt_state_get_req(
            usbdi_state_cb_t *cb);
```

5.18.3.2 Arguments

usbdi_state_cb_t ***cb***	Pointer to a control block that was allocated with *udi_cb_alloc()* given a cb_idx that was previously associated with USBDI_STATE_CB_NUM and given a LDD's pipe channel.

5.18.3.3 Description

usbdi_edpt_state_get_req() is called by the LDD to get the state of the endpoint associated with the pipe channel associated with the control block**.** In response to this call, the USBD issues a USB *GetStatus()*[11] request for the endpoint associated with ***pipe_channel***.

usbdi_edpt_state_get_req() may be used at any time, regardless of the state of the pipe or interface.

[11] See the Universal Serial Bus Specification for more information on the meaning of this term.

5.18.4 usbdi_edpt_state_get_ack_op_t

***usbdi_edpt_state_get_ack_op_t* – *typedef* for LDD supplied function called by the USBD on completion of a get endpoint state operation.**

5.18.4.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

typedef
void usbdi_edpt_state_get_ack_op_t(
            usbdi_state_cb_t *cb,
            udi_status_t status );
```

5.18.4.2 Arguments

usbdi_state_cb_t ****cb***	Pointer to a control block that was passed to *usbdi_pipe_state_get_req()*.
udi_status_t ***status***	See table below.

RETURNED STATUS	**DESCRIPTION**
UDI_OK	The endpoint state was successfully retrieved.
UDI_STAT_NOT_RESPONDING	The device is not responding or is inaccessible.

5.18.4.3 Description

The LDD shall supply a *usbdi_edpt_state_get_ack_op_t* function in its *usbdi_ldd_pipe_ops_t*. This function is called in response to the LDD's calling *usbdi_edpt_state_get_req()*. The *state* of CB will be either USBDI_STATE_ACTIVE or USBDI_STATE_HALTED. (See the Universal Serial Bus Specification for the meaning of the halted state).

The LDD shall either free the CB by calling *udi_cb_free()* or reuse the CB.

5.18.5 usbdi_edpt_state_get_ack_unused()

***usbdi_edpt_state_get_ack_unused()* - Proxy for *usbdi_edpt_state_get_req()*.**

5.18.5.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

usbdi_edpt_state_get_ack_op_t usbdi_edpt_state_get_ack_unused;
```

5.18.5.2 Description

usbdi_edpt_state_get_ack_unused() may be used in the *usbdi_ldd_pipe_ops_t* as the *usbdi_edpt_state_get_ack_op* if the LDD will never call *usbdi_edpt_state_get_req()*.

5.19 Getting and Setting Interface States

Setting the interface state causes all pipes associated with the interface (except for the default endpoint pipe) to move to that state. Once an interface is in the stalled or idle state, *usbdi_pipe_state_set_req()* cannot be used to change the state of the individual pipes. Only when the interface is in the active state may the pipe's state be changed with *usbdi_pipe_state_set_req()*.

The interface may be in one of the following states:

USBDI_STATE_ACTIVE	The pipes will accept transfer requests and send requests to the endpoint.
USBDI_STATE_STALLED	The pipes will accept transfer requests but not send requests to the endpoint.
USBDI_STATE_IDLE	The pipes will neither accept transfer requests nor send requests to the endpoint.

This section contains descriptions for the following functions and *typedefs*:

- *usbdi_intfc_state_set_req()*
- *usbdi_intfc_state_set_ack_op_t*
- *usbdi_intfc_state_set_ack_unused()*
- *usbdi_intfc_state_get_req()*
- *usbdi_intfc_state_get_ack_op_t*
- *usbdi_intfc_state_get_ack_unused()*

The following functions and *typedefs* are only used by the USBD (not by the LDD) and are not described in this specification. Callee-side interfaces may be derived from the caller-side interfaces by removing the first parameter, and vice versa. They are included in "Appendix A: Include File (open_usbdi.h)" for completeness:

- *usbdi_intfc_state_set_req_op_t*
- *usbdi_intfc_state_set_ack()*
- *usbdi_intfc_state_get_req_op_t*
- *usbdi_intfc_state_get_ack()*

5.19.1 usbdi_intfc_state_set_req()

***usbdi_intfc_state_set_req()* – Called by the LDD to set the state of an interface.**

5.19.1.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

void usbdi_intfc_state_set_req(
            usbdi_state_cb_t *cb );
```

5.19.1.2 Arguments

usbdi_state_cb_t ***cb***	Pointer to a control block that was allocated with *udi_cb_alloc()* given a cb_idx that was previously associated with USBDI_STATE_CB_NUM and given a LDD's interface channel.

5.19.1.3 Description

usbdi_intfc_state_set_req() is called by the LDD to change the state of the interface channel associated with the control block. The new interface state is set by the LDD in the *intfc_state* field of the CB. Valid values are USBDI_STATE_ACTIVE, USBDI_STATE_STALLED, and USBDI_STATE_IDLE. The result of setting the state of the interface is that all pipes controlled by the interface are moved to that state.

usbdi_intfc_state_set_req() may be used at anytime, regardless of the state of the pipes controlled by the interface.

5.19.2 usbdi_intfc_state_set_ack_op_t

***usbdi_intfc_state_set_ack_op_t – typedef* for LDD supplied function called by the USBD on completion of a set interface state operation.**

5.19.2.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

typedef
void usbdi_intfc_state_set_ack_op_t(
            usbdi_state_cb_t *cb,
            udi_status_t status );
```

5.19.2.2 Arguments

usbdi_state_cb_t ****cb***	Pointer to a control block that was passed to *usbdi_intfc_state_set_req()*.
udi_status_t ***status***	See table below.

RETURNED STATUS	**DESCRIPTION**
UDI_OK	The interface state was successfully set.
UDI_STAT_MISTAKEN_IDENTITY	The interface state was not set due to an invalid value in the CB.

5.19.2.3 Description

The LDD shall supply a *usbdi_intfc_state_set_ack_op_t* function in its *usbdi_ldd_intfc_ops_t*. This function is called in response to the LDD's calling *usbdi_intfc_state_set_req()*. Once the interface state has been set to anything other than USBDI_STATE_ACTIVE, pipes controlled by the interface may no longer be manipulated individually with *usbdi_pipe_state_set_req()*.

The LDD shall either free the CB by calling *udi_cb_free()* or reuse the CB.

5.19.3 usbdi_intfc_state_set_ack_unused()

***usbdi_intfc_state_set_ack_unused()* - Proxy for *usbdi_intfc_state_set_req()*.**

5.19.3.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

usbdi_intfc_state_set_ack_op_t usbdi_intfc_state_set_ack_unused;
```

5.19.3.2 Description

usbdi_intfc_state_set_ack_unused() may be used in the *usbdi_ldd_intfc_ops_t* as the *usbdi_intfc_state_set_ack_op* if the LDD will never call *usbdi_intfc_state_set_req()*.

5.19.4 usbdi_intfc_state_get_req()

***usbdi_intfc_state_get_req()* – Called by the LDD to retrieve the state of an interface.**

5.19.4.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

void usbdi_intfc_state_get_req(
            usbdi_state_cb_t *cb);
```

5.19.4.2 Arguments

usbdi_state_cb_t ***cb***	Pointer to a control block that was allocated with *udi_cb_alloc()* given a cb_idx that was previously associated with USBDI_STATE_CB_NUM and given a LDD's interface channel.

5.19.4.3 Description

usbdi_intfc_state_get_req() is called by the LDD to get the state of the interface channel associated with the control block. The state will be put in the *state* field of the control block. Valid values are USBDI_STATE_ACTIVE, USBDI_STATE_STALLED, and USBDI_STATE_IDLE.

usbdi_intfc_state_get_req() may be used at any time, regardless of the state of the pipes controlled by the interface.

5.19.5 usbdi_intfc_state_get_ack_op_t

***usbdi_intfc_state_get_ack_op_t – typedef* for LDD supplied function called by the USBD on completion of a get interface state operation.**

5.19.5.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

typedef
void usbdi_intfc_state_get_ack_op_t(
            usbdi_state_cb_t *cb );
```

5.19.5.2 Arguments

usbdi_state_cb_t ***cb***	Pointer to a control block that was passed to *usbdi_intfc_state_get_req()*.

5.19.5.3 Description

The LDD shall supply a *usbdi_intfc_state_get_ack_op_t* function in its *usbdi_ldd_intfc_ops_t*. This function is called in response to the LDD's calling *usbdi_intfc_state_get_req()*. The *state* field of the CB will be one of the following values: USBDI_STATE_ACTIVE, USBDI_STATE_STALLED, or USBDI_STATE_IDLE.

The LDD shall either free the CB by calling *udi_cb_free()* or reuse the CB.

5.19.6 usbdi_intfc_state_get_ack_unused()

***usbdi_intfc_state_get_ack_unused()* - Proxy for *usbdi_intfc_state_get_req()*.**

5.19.6.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

usbdi_intfc_state_get_ack_op_t usbdi_intfc_state_get_ack_unused;
```

5.19.6.2 Description

usbdi_intfc_state_get_ack_unused() may be used in the *usbdi_ldd_intfc_ops_t* as the *usbdi_intfc_state_get_ack_op* if the LDD will never call *usbdi_intfc_state_get_req()*.

5.20 Getting Device States

At times an LDD may need to determine what state its associated USB device is in. USB devices support the following LDD visible states. (These states are described in detail in the Universal Serial Bus Specification.)

USBDI_STATE_CONFIGURED	The device has been configured.
USBDI_STATE_SUSPENDED	The device is in the suspended state.

This section contains descriptions for the following functions and *typedef*:

- *usbdi_device_state_get_req()*
- *usbdi_device_state_get_ack_op_t*
- *usbdi_device_state_get_ack_unused()*

The following function and *typedef* are used only by the USBD (not by the LDD) and are not described in this specification. Callee-side interfaces may be derived from the caller-side interfaces by removing the first parameter, and vice versa. They are included in "Appendix A: Include File (open_usbdi.h)" for completeness:

- *usbdi_device_state_get_req_op_t*
- *usbdi_device_state_get_ack()*

5.20.1 usbdi_device_state_get_req()

***usbdi_device_state_get_req()* – Called by the LDD to retrieve the state of a device.**

5.20.1.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

void usbdi_device_state_get_req(
            usbdi_state_cb_t *cb);
```

5.20.1.2 Arguments

usbdi_state_cb_t ***cb***	Pointer to a control block that was allocated with *udi_cb_alloc()* given a cb_idx that was previously associated with USBDI_STATE_CB_NUM and given the LDD's bind channel.

5.20.1.3 Description

usbdi_device_state_get_req() is called by the LDD to get the state of the device associated with the LDD's bind channel**.** Valid values are combinations of the following: USBDI_STATE_CONFIGURED and USBDI_STATE_SUSPENDED. Refer to the Universal Serial Bus Specification for a discussion of these states.

5.20.2 usbdi_device_state_get_ack_op_t

***usbdi_device_state_get_ack_op_t – typedef* for LDD supplied function called by the USBD on completion of a get device state operation.**

5.20.2.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

typedef
void usbdi_device_state_get_ack_op_t(
            usbdi_state_cb_t *cb );
```

5.20.2.2 Arguments

usbdi_state_cb_t ****cb***	Pointer to a control block that was passed to *usbdi_device_state_get_req()*.

5.20.2.3 Description

The LDD shall supply a *usbdi_device_state_get_ack_op_t* function in its *usbdi_ldd_intfc_ops_t* if the LDD may call *usbdi_device_state_get_req()*. This function is called in response to the LDD's calling *usbdi_device_state_get_req()*

The LDD shall either free the CB by calling *udi_cb_free()* or reuse the CB.

5.20.3 usbdi_device_state_get_ack_unused()

***usbdi_device_state_get_ack_unused()* - Proxy for *usbdi_device_state_get_req()*.**

5.20.3.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

usbdi_device_state_get_ack_op_t usbdi_device_state_get_ack_unused;
```

5.20.3.2 Description

usbdi_device_state_get_ack_unused() may be used in the *usbdi_ldd_intfc_ops_t* as the *usbdi_device_state_get_ack_op* if the LDD will never call *usbdi_device_state_get_req()*.

5.21 Retrieving Descriptors

This section describes how LDDs may retrieve USB descriptors. The following functions allow the USBD to manage the retrieval of descriptors in an implementation-specific way. Some USBD implementations may choose to cache the configuration descriptors while others may choose to read the descriptor from the device as needed.

This mechanism allows for the retrieval of descriptors returned by the device as individual descriptors or descriptors from within the configuration descriptor, such as the interface descriptor.

The USBD allocates the needed memory to create a copy of the requested descriptor (as a movable memory block). It is the responsibility of the LDD to release this memory by calling *udi_mem_free()*.

This section contains descriptions for the following functions and *typedefs*:

- *usbdi_desc_cb_t*
- *usbdi_desc_req()*
- *usbdi_desc_ack_op_t*
- *usbdi_desc_ack_unused()*

The following function and *typedef* are used only by the USBD (not by the LDD) and are not described in this specification. Callee-side interfaces may be derived from the caller-side interfaces by removing the first parameter, and vice versa. They are included in “Appendix A: Include File (open_usbdi.h)” for completeness:

- *usbdi_desc_req_op_t*
- *usbdi_desc_ack()*

5.21.1 usbdi_desc_cb_t

usbdi_desc_cb_t **– Used in retrieving descriptors from logical USB devices.**

5.21.1.1 Synopsis

```
typedef struct {

      udi_cb_t gcb;           /* UDI generic control block */

/* Type of descriptor to retrieve.  May be one of the following
 * or any other valid descriptor type.
 */
      udi_ubit8_t desc_type;
#define USB_DESC_TYPE_DEVICE        0x01
#define USB_DESC_TYPE_CONFIG        0x02
#define USB_DESC_TYPE_STRING        0x03
#define USB_DESC_TYPE_INTFC         0x04
#define USB_DESC_TYPE_EDPT          0x05
      udi_ubit8_t desc_index; /* Index of descriptor to retrieve */
      udi_ubit16_t desc_ID;   /* Language ID for string descriptors,
                               * Logical-Device ID for all other
                               * descriptors.  See the Common Class
                               * Logical-Devices Feature Specification
                               * for more info on Logical-Device Ids.
                               */
      udi_ubit16_t desc_length;     /* # of bytes to retrieve */
      udi_buf_t *desc_buf;          /* Buffer containing returned
                                       descriptor */
} usbdi_desc_cb_t;

#define USBDI_DESC_CB_NUM           5
```

5.21.1.2 Description

The descriptor request control block is used in OpenUSBDI operations that involve retreiving USB logical device descriptors.

In order to use this type of control block it must be associated with a control block index by including `USBDI_DESC_CB_NUM` in a `udi_cb_init_t` in the driver's `udi_init_info`.

5.21.2 usbdi_desc_req()

***usbdi_desc_req()* – Called by the LDD to retrieve a copy of a USB descriptor.**

5.21.2.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

void usbdi_desc_req(
            usbdi_desc_cb_t *cb );
```

5.21.2.2 Arguments

usbdi_desc_cb_t ***cb**	Pointer to a control block that was allocated with *udi_cb_alloc()* given a cb_idx that was previously associated with USBDI_DESC_CB_NUM and given a LDD's interface channel.

5.21.2.3 Description

usbdi_desc_req() is called by the LDD to retrieve any descriptor from the device. Alternatively, LDDs may retrieve device descriptors by setting up a *GetDescriptor()*[12] request and sending it to the default pipe of the interface with *usbdi_control_xfer_req().*

The *desc_type* of CB may be device, configuration, string, interface, endpoint, USB class-specific, or vendor specific.

The *desc_index* of CB is zero based. When retrieving descriptors contained in the configuration descriptor, *desc_index* refers to the instance of the descriptor for the interface related to ***intfc_channel***. For example, to retrieve the first endpoint descriptor of an interface, the *desc_index* shall be zero ("0") even if the interface is not the first of the configuration.

The *desc_length* of CB shall be set to the length, in bytes, of the descriptor. The *desc_buf* of CB shall be set to NULL.

The USB configuration descriptors may be retrieved in part or in their entirety (all descriptors contained in the configuration) by setting the *desc_length* appropriately.

This operation is performed on an interface channel.

[12] See the Universal Serial Bus Specification for more information on *GetDescriptor()*.

5.21.3 usbdi_desc_ack_op_t

***usbdi_desc_ack_op_t* – *typedef* for LDD supplied function called by the USBD on completion of a get descriptor operation.**

5.21.3.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

typedef
void usbdi_desc_ack_op_t(
            usbdi_desc_cb_t *cb,
            udi_status_t status );
```

5.21.3.2 Arguments

usbdi_desc_cb_t ****cb***	Pointer to a control block that was passed to *usbdi_desc_req()*.
udi_status_t ***status***	See table below.

RETURNED STATUS	DESCRIPTION
UDI_OK	The descriptor was successfully retrieved. desc_buf points to a valid buffer.
UDI_STAT_NOT_RESPONDING	The device is not responding or is inaccessible.
UDI_STAT_MISTAKEN_IDENTITY	The CB was set up with an invalid argument. This is the case if desc_type or desc_index are invalid for the device.

5.21.3.3 Description

The LDD shall supply a *usbdi_desc_ack_op_t* function in its *usbdi_ldd_intfc_ops_t*. This function is called in response to the LDD's calling *usbdi_desc_req()*. If the descriptor was successfully retrieved, ***status*** will be UDI_OK and *desc_buf* of CB shall point to a buffer containing the descriptor. The *desc_length* of CB will contain the number of bytes that are valid in *desc_buf*. The LDD owns the buffer referred to by *desc_buf* and is responsible for freeing it by calling *udi_buf_free()*.

The LDD shall either free the CB by calling *udi_cb_free()* or reuse the CB for another call to *usbdi_desc_req()*.

5.21.4 usbdi_desc_ack_unused()

***usbdi_desc_ack_unused()* - Proxy for *usbdi_desc_req()*.**

5.21.4.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

usbdi_desc_ack_op_t usbdi_desc_ack_unused;
```

5.21.4.2 Description

usbdi_desc_ack_unused() may be used in the *usbdi_ldd_intfc_ops_t* as the *usbdi_desc_ack_op* if the LDD will never call *usbdi_desc_req()*.

5.22 Changing a Device's Configuration

This section contains descriptions for the following functions and *typedef*:

- *usbdi_config_set_req()*
- *usbdi_config_set_ack_op_t*
- *usbdi_config_set_ack_unused()*

The following function and *typedef* are used only by the USBD (not by the LDD) and are not described in this specification. Callee-side interfaces may be derived from the caller-side interfaces by removing the first parameter, and vice versa. They are included in "Appendix A: Include File (open_usbdi.h)" for completeness:

- *usbdi_config_set_req_op_t*
- *usbdi_config_set_ack()*

5.22.1 usbdi_config_set_req()

usbdi_config_set_req() **– Called by the LDD to set the configuration of a USB device.**

5.22.1.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

void usbdi_config_set_req(
            usbdi_misc_cb_t *cb,
            udi_ubit16_t config_value);
```

5.22.1.2 Arguments

usbdi_misc_cb_t ****cb***	Pointer to a control block allocated with *udi_cb_alloc()* given a cb_idx that was previously associated with USBDI_MISC_CB_NUM and given a LDD's interface channel.
udi_ubit16_t **config_value**	New configuration value.

5.22.1.3 Description

usbdi_config_set_req() allows LDDs to set the configuration of the device. This is the only way that the configuration may be changed. The LDD shall not set the configuration by issuing a *SetConfiguration()*[13] device request to the default pipe, the USBD shall fail this request.

LDDs are not required to, and in most cases shall not, perform a *usbdi_config_set_req()*. The USBD shall set the device's configuration before the LDDs are bound. There may be some LDDs that do need to perform a *usbdi_config_set_req()* (such as some vendor unique LDDs), but in most cases it is not necessary.

All interfaces contained by the current configuration shall be closed before *usbdi_config_set_req()* may be performed. The new configuration value is set by the LDD in ***config_value*** and shall be a valid configuration for the device.

An LDD may determine the current configuration value by issuing a *GetConfiguration()* device request to the default pipe channel.

This operation is performed on an interface channel.

[13] See the Universal Serial Bus Specification for more information on SetConfiguration().

5.22.2 usbdi_config_set_ack_op_t

***usbdi_config_set_ack_op_t – typedef* for LDD supplied function called by the USBD on completion of a set configuration operation.**

5.22.2.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

typedef
void usbdi_config_set_ack_op_t(
            usbdi_misc_cb_t *cb,
            udi_status_t status );
```

5.22.2.2 Arguments

usbdi_desc_cb_t ****cb***	Pointer to a control block that was passed to *usbdi_desc_req()*.
udi_status_t ***status***	See table below.

RETURNED STATUS	**DESCRIPTION**
UDI_OK	The configuration of the device was successfully changed.
UDI_STAT_INVALID_STATE	One or more interfaces associated with the device are opened.
UDI_STAT_NOT_RESPONDING	The device is not responding or is inaccessible.
UDI_STAT_MISTAKEN_IDENTITY	The config_value given to usbdi_config_set_req() is invalid.

5.22.2.3 Description

The LDD shall supply a *usbdi_config_set_ack_op_t* function in its *usbdi_ldd_intfc_ops_t*. This function is called in response to the LDD’s calling *usbdi_config_set_req()*. If the configuration was successfully changed, ***status*** will be UDI_OK.

Once the configuration has been changed, all LDDs bound to the previous configuration will be unbound (via the UDI Management Agent) and the new configuration will be bound.

The LDD shall either free the CB by calling *udi_cb_free()* or reuse the CB.

5.22.3 usbdi_config_set_ack_unused()

***usbdi_config_set_ack_unused()* - Proxy for *usbdi_config_set_req()*.**

5.22.3.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

usbdi_config_set_ack_op_t usbdi_config_set_ack_unused;
```

5.22.3.2 Description

usbdi_config_set_ack_unused() may be used in the *usbdi_ldd_intfc_ops_t* as the *usbdi_config_set_ack_op* if the LDD will never call *usbdi_config_set_req()*.

5.23 Asynchronous Events

Asynchronous event notification is provided to notify LDDs of events they may not have initiated. These events include:

Device suspended (USBDI_ASYNC_SUSPEND)	The USB device associated with the interface managed by the LDD has been suspended. The state of the interface has been moved to USBDI_STATE_STALLED (see Section 5.18.5, "Getting and Setting Interface States") by the USBD.
Device wakeup (USBDI_ASYNC_WAKEUP)	The USB device associated with the interface managed by the LDD is no longer suspended. Once the LDD calls *usbdi_async_notify_ack()*, the state of the interface will be moved to USBDI_STATE_ACTIVE by the USBD.
USB bandwidth needed (USBDI_ASYNC_BANDWIDTH_NEEDED)	A USB device can't be configured because there is not enough available bandwidth. The LDD may attempt to change to an alternate interface that requires less USB bandwidth.

All LDDs that have opened interfaces with the USBDI_INTFC_OPEN_ASYNC_EVENT flag set will be notified when the device has been suspended or awoken. All LDDs with alternate interfaces will be notified when USB bandwidth is needed. A well-behaved LDD that can reduce its USB bandwidth consumption by switching to an alternate interface may do so when this asynchronous event is received and before it calls *usbdi_async_event_res()*.

This section contains descriptions for the following functions and *typedef*:

- *usbdi_async_event_ind_op_t*
- *usbdi_async_event_ind_unused()*
- *usbdi_async_event_res()*

The following function and *typedef* are used only by the USBD (not by the LDD) and are not described in this specification. Callee-side interfaces may be derived from the caller-side interfaces by removing the first parameter, and vice versa. They are included in "Appendix A: Include File (open_usbdi.h)" for completeness:

- *usbdi_async_event_ind()*
- *usbdi_async_event_res_op_t*

5.23.1 usbdi_async_event_ind_op_t

***usbdi_async_event_ind_op_t – typedef* for LDD supplied function called by the USBD to notify the LDD of an asynchronous event.**

5.23.1.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

typedef
void usbdi_async_event_ind_op_t(
            usbdi_misc_cb_t *cb,
            udi_ubit16_t async_event);
```

5.23.1.2 Arguments

usbdi_misc_cb_t ****cb***	Pointer to a control block allocated and set up by the USBD.
udi_ubit16_t ***async_event***	Asynchronous event that has occurred. May be one of the following: USBDI_ASYNC_SUSPEND, USBDI_ASYNC_WAKEUP, USBDI_ASYNC_BANDWIDTH_NEEDED.

5.23.1.3 Description

The LDD shall supply a *usbdi_async_event_ind_op_t* function in its *usbdi_ldd_intfc_ops_t*. This function is called in response to an asynchronous event detected by the USBD. All interfaces that were opened with the USBDI_INTFC_OPEN_ASYNC_EVENT flag set will be notified of asynchronous events. LDDs that control more than one interface will receive notifications for each interface opened with the USBDI_INTFC_OPEN_ASYNC_EVENT flag set.

The CB was allocated by the USBD and is returned to the USBD via *usbdi_async_event_res()* . This CB shall not be modified or deallocated by the LDD.

5.23.2 usbdi_async_event_ind_unused()

***usbdi_async_event_ind_unused()* - Proxy for *usbdi_async_event_req*.**

5.23.2.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

usbdi_async_event_ind_op_t usbdi_async_event_ind_unused;
```

5.23.2.2 Description

usbdi_async_event_ind_unused() may be used in the *usbdi_ldd_intfc_ops_t* as the *usbdi_async_event_ind_op* if the LDD did not set the USBDI_INTFC_OPEN_ASYNC_EVENT flag in *usbdi_intfc_op_req()* and thus expects never to receive a *usbdi_async_event_ind*.

5.23.3 usbdi_async_event_res()

***usbdi_async_event_res()* – Called by the LDD in response to the USBD calling the LDD's *usbdi_async_event_ind_op_t* function.**

5.23.3.1 Synopsis

```
#include <udi.h>
#include <open_usbdi.h>

void usbdi_async_event_res(
            usbdi_misc_cb_t *cb );
```

5.23.3.2 Arguments

usbdi_misc_cb_t ****cb***	Pointer to a control block passed to *usbdi_async_event_ind_op_t*.

5.23.3.3 Description

usbdi_async_event_res() is called by the LDD in response to the USBD's calling the LDD's *usbdi_async_event_ind_op_t* function. The LDD shall call this operation to acknowledge receipt of the asynchronous event.

After this operation is called by the LDD, the state of the CB is unspecified.

This operation is performed on an interface channel.

6 Helpful Hints

This section contains various suggestions that may aid in the development of OpenUSBDI LDDs. Refer to the sample driver in Appendix B to help clarify the usage of the OpenUSBDI functions and structures.

1. LDDs shall provide function pointers for all *usbdi_ldd_intfc_ops_t* and *usbdi_ldd_pipe_ops_t*. Most LDDs will not use all of the functions that shall be provided. For example, a *usbdi_isoc_xfer_ack_op_t* is not needed by a LDD that doesn't control an isochronous endpoint. In this case, the LDD may use the "unused" function (described by this document) specific to the operation in place of an LDD unique function.

2. LDDs may achieve better performance by having more than one transfer CB outstanding to a pipe channel at a time. This allows the LDD to process a completing CB while another CB is being processed by the host controller.

7 Appendix A: Include File (open_usbdi.h)

```
#ifndef _OPEN_USBDI_H_
#define _OPEN_USBDI_H_

#if OPEN_USBDI_VERSION == 0x0+100

/*
 * Unless otherwise specified, the following abbreviations will be used:
 *
 * ldd         - logical-device driver
 * cb           - Control Block
 * _call        - Callback Function
 * n_           - Number of ...
 * _t           - Type
 * ops         - UDI channel operation vectors
 * xfer        - transfer
 * edpt        - endpoint
 * intfc       - interface
 * desc        - descriptor
 * config      - configuration
 */

/*
 * The following structs and defines are based on the USB
 * core spec, rev 1.1
 */

/*
 * USB Device Requests - These requests are made using control transfers.
 * The request and the request's parameters are sent to the
 * device in the set up packet.  The host is responsible for establishing
 * the values passed in the following fields.  Every set up packet has eight
 * bytes.
 */
typedef struct  {
    udi_ubit8_t bmRequestType; /* Characteristics of request */
#define USB_DEVICE_REQUEST_TYPE_DEVICE_RECIPIENT     0x00
#define USB_DEVICE_REQUEST_TYPE_INTERFACE_RECIPIENT  0x01
#define USB_DEVICE_REQUEST_TYPE_ENDPOINT_RECIPIENT   0x02
#define USB_DEVICE_REQUEST_TYPE_OTHER_RECIPIENT      0x03
#define USB_DEVICE_REQUEST_TYPE_STANDARD_TYPE        0x00
#define USB_DEVICE_REQUEST_TYPE_CLASS_TYPE           0x20
#define USB_DEVICE_REQUEST_TYPE_VENDOR_TYPE          0x40
#define USB_DEVICE_REQUEST_TYPE_HOST_TO_DEVICE       0x00
#define USB_DEVICE_REQUEST_TYPE_DEVICE_TO_HOST       0x80

    udi_ubit8_t bRequest;  /* Specific request */
#define USB_DEVICE_REQUEST_GET_STATUS        0
#define USB_DEVICE_REQUEST_CLEAR_FEATURE     1
#define USB_DEVICE_REQUEST_GET_STATE         2 /* Reserved for future use?*/
#define USB_DEVICE_REQUEST_SET_FEATURE       3
#define USB_DEVICE_REQUEST_SET_ADDRESS       5
#define USB_DEVICE_REQUEST_GET_DESCRIPTOR    6
#define USB_DEVICE_REQUEST_SET_DESCRIPTOR    7
#define USB_DEVICE_REQUEST_GET_CONFIGURATION 8
#define USB_DEVICE_REQUEST_SET_CONFIGURATION 9
#define USB_DEVICE_REQUEST_GET_INTERFACE     10
#define USB_DEVICE_REQUEST_SET_INTERFACE     11
#define USB_DEVICE_REQUEST_SYNCH_FRAME       12

    udi_ubit8_t wValue0;
    udi_ubit8_t wValue1;      /* Word-sized field that varies
                               * according to request
                               */
#define USB_DEVICE_REQUEST_DEVICE_REMOTE_WAKEUP      1
#define USB_DEVICE_REQUEST_ENDPOINT_HALT     0

    udi_ubit8_t wIndex0;
    udi_ubit8_t wIndex1;      /* Word sized field that varies according to
                               * request; typically used to pass an index
                               * or offset
                               */
    udi_ubit8_t wLength0;
    udi_ubit8_t wLength1;     /* Number of bytes to transfer if there is a
                               * data phase
                               */
} usb_device_request_t;

/*
 * Descriptor Types
 */
#define USB_DESCRIPTOR_TYPE_DEVICE           0x01
#define USB_DESCRIPTOR_TYPE_CONFIGURATION    0x02
#define USB_DESCRIPTOR_TYPE_STRING           0x03
#define USB_DESCRIPTOR_TYPE_INTERFACE        0x04
#define USB_DESCRIPTOR_TYPE_ENDPOINT         0x05

/*
 * Device Descriptor - A device descriptor describes general information
 * about a USB device.  It includes information that applies globally to
 * the device and all of the device's configurations.  A USB device has
 * only one device descriptor.
 *
 * All USB devices have an endpoint zero used by the default pipe.  The
 * maximum packet size of a device's endpoint zero is described in the device
 * descriptor.  Endpoints specific to a configuration and its interface(s)
 * are described in the configuration descriptor.  A configuration and its
 * interface(s) do not include an endpoint descriptor for endpoint zero.
 * Other than the maximum packet size, the characteristics of endpoint zero
 * are defined by the USB specification and are the same for all USB devices.
 */
struct usb_device_descriptor {
    udi_ubit8_t bLength;      /* Numeric expression specifying the size of
                               * this descriptor
                               */
    udi_ubit8_t bDescriptorType;/* Device descriptor type (assigned by USB) */

    udi_ubit8_t bcdUSB0;
    udi_ubit8_t bcdUSB1;      /* binary-coded decimal specification # */
    udi_ubit8_t bDeviceClass; /* Class code (assigned by USB).  Note that
                               * the HID class is defined in the Interface
                               * descriptor
                               */
    udi_ubit8_t bDeviceSubClass;/* Subclass code (assigned by USB).  These
                               * codes are qualified by the value of the
                               * DeviceClass field.
                               */
    udi_ubit8_t bDeviceProtocol;/* Protocol code.  These codes are qualified
                               * by the value of the DeviceSubClass field.
                               */
    udi_ubit8_t bMaxPacketSize; /* Maximum packet size for endpoint zero (only
                               * 8, 16, 32, or 4 are valid).
                               */
    udi_ubit8_t idVendor0;
    udi_ubit8_t idVendor1;    /* Vendor ID (assigned by USB) */

    udi_ubit8_t idProduct0;
    udi_ubit8_t idProduct1;   /* Product ID (assigned by manufacturer) */

    udi_ubit8_t bcdDevice0;
    udi_ubit8_t bcdDevice1;   /* Device release number */

    udi_ubit8_t iManufacturer;       /* Index of String desc describing manufacture */
    udi_ubit8_t iProduct;     /* Index of string desc describing product */
    udi_ubit8_t iSerialNumber;       /* Index of String desc describing serial # */
    udi_ubit8_t bNumConfigurations;  /* Number of possible configurations */
};

/*
```

```
 * Configuration Descriptor - The configuration descriptor describes
 * information about a specific device configuration.  The descriptor
 * contains a ConfigurationValue field with a value that, when used as
 * a parameter to the Set Configuration request, causes the device to
 * assume the described configuration.
 *
 * The descriptor describes the number of interfaces provided by the
 * configuration.  Each interface may operate independently.  For example,
 * an ISDN device might be configured with two interfaces, each providing
 * 64kBs bi-directional channels that have separate data sources or sinks
 * on the host.  Another configuration might present the ISDN device as
 * a single interface, bonding the two channels into one 128 kBs
 * bi-directional channel.
 *
 * When the host requests the configuration descriptor, all related
 * interface and endpoint descriptors are returned.
 *
 * A USB device has one or more configuration descriptors.  Each
 * configuration has one or more interfaces and each interface has one or
 * more endpoints.  An endpoint is not shared among interfaces within a
 * single configuration unless the endpoint is used by alternate settings
 * of the same interface.  Endpoints may be shared among interfaces that are
 * part of different configurations without this restriction.
 *
 * Once configured, devices may support limited adjustments to the
 * configuration.  If a particular interface has alternate settings, an
 * alternate may be selected after configuration.  Within an interface,
 * an isochronous endpoint's maximum packet size may also be adjusted.
 */
struct usb_configuration_descriptor {
    udi_ubit8_t bLength;      /* Size of this descriptor in bytes */
    udi_ubit8_t bDescriptorType;/* Configuration (assigned by USB) */

    udi_ubit8_t wTotalLength0;
    udi_ubit8_t wTotalLength1;  /* Total length of data returned for this
                               * configuration.  Includes the combined
                               * length of all returned descriptors
                               * (configuration, interface, endpoint, and
                               * HID) returned for this configuration.  This
                               * value includes the HID descriptor but none
                               * of the other HID class descriptors (report
                               * or designator).
                               */
    udi_ubit8_t bNumInterfaces; /* Number of interfaces supported by this
                               * configuration.
                               */
    udi_ubit8_t bConfigurationValue;  /* Value to use as an argument to Set
                                     * Configuration to select this configuration
                                     */
    udi_ubit8_t iConfiguration; /* Index of string descriptor describing this
                               * configuration.
                               */
    udi_ubit8_t bmAttributes; /* Configuration characteristics */
#define USB_CONFIG_BUS_POWERED   0x80
#define USB_CONFIG_SELF_POWERED  0x40
#define USB_CONFIG_REMOTE_WAKEUP 0x20

    udi_ubit8_t MaxPower;     /* Maximum power consumption of USB device
                               * from bus in this specific configuration
                               * when the device is fully operational.
                               * Expressed in 2mA units -- for example,
                               * 50 = 100mA.
                               */
};

/*
 * Interface Descriptor - This descriptor describes a specific interface
 * provided by the associated configuration.  A configuration provides one
 * or more interfaces, each with its own endpoint descriptors describing
 * a unique set of endpoints within the configuration.  When a configuration
 * supports more than one interface, the endpoints for a particular interface
 * immediately follow the interface descriptor in the data returned by the
 * Get Configuration request.  An interface descriptor is always returned
 * as part of a configuration descriptor.  It cannot be directly accessed
 * with a Get or Set Descriptor request.
 *
 * An interface may include alternate settings that allow the endpoints and/or
 * their characteristics to be varied after the device has been configured.
 * The default setting for an interface is always alternate setting zero.
 * The Set Interface request is used to select an alternate setting or to
 * return to the default setting .  The Get Interface request returns the
 * selected alternate setting.
 *
 * Alternate settings allow a portion of the device configuration to be
 * varied while other interfaces remain in operation.  If a configuration
 * has alternate settings for one or more of its interfaces, a separate
 * interface descriptor and its associated endpoints are included for each
 * setting.
 *
 * If a device configuration supported a single interface with two alternate
 * settings, the configuration descriptor would be followed by an interface
 * descriptor with the bIntefaceNumber and bAlternaSetting fields set to zero
 * and then the endpoint descriptors for that setting, followed by another
 * interface descriptor and its associated endpoint descriptors.  The second
 * interface descriptor's InterfaceNumber field would also be set to zero,
 * but the AlternateSetting field of the second interface descriptor would
 * be set to one.
 *
 * If an interface only uses endpoint zero, no endpoint descriptors follow
 * the interface descriptor and the interface identifies a request interface
 * that uses the default pipe attached to endpoint zero.  In this case the
 * NumEndpoints field shall be set to zero.
 *
 * An interface descriptor never includes endpoint zero in the number of
 * endpoints.
 */
struct usb_interface_descriptor {
    udi_ubit8_t bLength;        /* Size of this descriptor in bytes */
    udi_ubit8_t bDescriptorType;  /* Interface descriptor type */
    udi_ubit8_t bInterfaceNumber; /* Number of interface.  Zero-based value
                                 * identifying the index in the array of
                                 * concurrent interfaces supported by this
                                 * configuration
                                 */
    udi_ubit8_t bAlternateSetting;  /* Value used to select alternate setting
                                   * for the interface identified in the prior
                                   * field.
                                   */
    udi_ubit8_t bNumEndpoints;       /* Number of endpoints used by this interface
                               * (excluding endpoint zero).  If this value
                               * is zero, this interface only uses endpoint
                               * zero.
                               */
    udi_ubit8_t bInterfaceClass;    /* Class code */
    udi_ubit8_t bInterfaceSubClass; /* Subclass code */
#define USB_INTERFACE_SUBCLASS_NO_SUBCLASS    0x00
#define USB_INTERFACE_SUBCLASS_BOOT_INTERFACE 0x01

    udi_ubit8_t bInterfaceProtocol; /* Protocol code */
#define USB_INTERFACE_PROTOCOL_NONE 0x00

    udi_ubit8_t iInterface;   /* Index of string descriptor describing this
                               * interface
                               */
};

/*
 * Endpoint Descriptor - Each endpoint used for an interface has its own
 * descriptor.  This descriptor contains the information required by the
 * host to determine the bandwidth requirements of each endpoint.  An endpoint
 * descriptor is always returned as part of a configuration descriptor.  It
 * cannot be directly accessed with a Get or Set Descriptor request.  There
 * is never an endpoint descriptor for endpoint zero.
 */
struct usb_endpoint_descriptor {
    udi_ubit8_t bLength;      /* Size of this descriptor in bytes */
    udi_ubit8_t bDescriptorType;/* Endpoint descriptor type (assigned by USB)*/
    udi_ubit8_t bEndpointAddress;/* The address of the endpoint on the USB
                               * device described by this descriptor.  The
                               * address is encoded as follows:
                               *    Bit 0..3  The endpoint number
                               *    Bit 4..6  Reserved, reset to zero
                               *    Bit 7     Direction, ignored for Control
                               *              endpoints: 0 - OUT endpoint,
                               *              1 - IN endpoint
                               */
    udi_ubit8_t bmAttributes; /* This field describes the endpoint's
                               * attributes when it is configured using the
                               * Configuration Value.
                               *    Bit 0..1  Transfer type:
                               *    00        Control
                               *    01        Isochronous
                               *    10        Bulk
                               *    11        Interrupt
                               *    All other bits are reserved
                               */
#define USB_ENDPOINT_CONTROL         0
#define USB_ENDPOINT_ISOCHRONOUS     1
#define USB_ENDPOINT_BULK            2
#define USB_ENDPOINT_INTERRUPT       3
#define USB_ENDPOINT_ATTRIBUTE_MASK  3

    udi_ubit8_t bMaxPacketSize_lo;
    udi_ubit8_t bMaxPacketSize_hi;   /* Maximum packet size this endpoint is
                                       * capable of sending or receiving when this
                                       * configuration is selected.  For interrupt
                                       * endpoints, this value is used to reserve
                                       * the bus time in the schedule, required for
                                       * the per frame data payloads.  Smaller data
                                       * payloads may be sent, but will terminate
                                       * the transfer and thus require intervention
                                       * to restart.
                                       */
    udi_ubit8_t bInterval;            /* Interval for polling endpoint for data
                                       * transfers.  Expressed in milliseconds.
                                       */
};

/*
 * String Descriptor - String descriptors are optional.  If a device does
 * not support string descriptors, all references to string descriptors
 * within device, configuration, and interface descriptors shall be reset
 * to zero.
 *
 * String descriptors use UNICODE encodings as defined by The Unicode
 * Standard, Worldwide Character Encoding, Version 1.0, Volumes 1 and 2.
 * The strings in a USB device may support multiple languages.  When
 * requesting a string descriptor, the requester specifies the desired
 * language using a sixteen-bit language ID.  String index 0 for all
 * languages returns an array of two-byte codes supported by the device.
 * A USB device may omit all string descriptors.
 *
 * The UNICODE string descriptor is not NULL terminated.  The string
 * length is computed by subtracting two from the value of the first byte
 * of the descriptor.
 */
struct usb_string_descriptor {
    udi_ubit8_t Length;              /* Length of descriptor in bytes */
    udi_ubit8_t DescriptorType;      /* Descriptor Type */
    udi_ubit8_t String[1];
};

/*
```

```
 * USBDI Status Return Values
 */
#define USBDI_STAT_NOT_ENOUGH_BANDWIDTH      (UDI_STAT_META_SPECIFIC|1)
#define USBDI_STAT_STALL                     (UDI_STAT_META_SPECIFIC|2)

/*
 * Start OpenUSBDI Metalanguage
 *
 * For each OpenUSBDI Metalanguage operation there is an associated set of
 * functional interface declarations (the utility function
 * usbdi_frame_distrib() is not considered an 'operation').  These functions
 * define how the LDD (logical device driver) and the USBD (USB protocol
 * driver) communicate.  The following five functions are provided for each
 * metalanguage operation:
 *
 * usbdi_XX_req
 *     Called by the LDD to send the CB to the USBD.
 *
 * usbdi_XX_req_op_t
 *     Function type for the USBD supplied function which will
 *     be called in response to the LDD performing the
 *     "usbdi_XX_req" call.  This function is NOT called by the LDD.
 *
 * usbdi_XX_ack
 *     Called by the USBD on completion of the CB.  This function
 *     is NOT called by the LDD.
 *
 * usbdi_XX_ack_op_t
 *     Function type for the LDD supplied function which will be
 *     called on completion of the CB.
 *
 * Summary:
 *
 *     To perform a request the LDD allocates a CB by calling
 *     udi_cb_alloc() with a CB specific index argument.  This
 *     is the same index value which was given to usbdi_XX_cb_init()
 *     at driver init time.
 *
 *     The LDD sets up this CB and sends it to the USBD by calling
 *     usbdi_XX_req().
 *
 *     The LDD shall provide a usbdi_XX_ack_op_t function which
 *     will be called when the CB completes.
 *
 *     usbdi_XX_req_op_t and usbdi_XX_ack() are provided for
 *     completeness and are used by the USBD (not the LDD).
 */

/*
 * USBDI Miscellaneous Control Block
 *
 * The usbdi_misc_cb_t control block is used for the majority
 * of the OpenUSBDI operations.
 */
typedef struct {
  udi_cb_t gcb;
} usbdi_misc_cb_t;

#define USBDI_MISC_CB_NUM     1


/*
 * Functional Device Driver binding
 *
 * This CB allows a LDD to be bound to a USB interface channel.  This
 * binding shall be performed for each interface that a LDD controls.
 */

void           usbdi_bind_req(       usbdi_misc_cb_t *cb);

typedef void   usbdi_bind_req_op_t(   usbdi_misc_cb_t *cb);
```

```
void           usbdi_bind_ack(       usbdi_misc_cb_t *cb,
                                     udi_index_t n_intfc,
                                     udi_status_t status);

typedef void   usbdi_bind_ack_op_t(  usbdi_misc_cb_t *cb,
                                     udi_index_t n_intfc,
                                     udi_status_t status);

/*
 * Functional Device Driver unbinding
 *
 * This CB allows a LDD to be unbound from a USB interface channel.  This
 * unbinding shall be performed for each interface that a LDD controls.
 */

void           usbdi_unbind_req(     usbdi_misc_cb_t *cb);

typedef void   usbdi_unbind_req_op_t( usbdi_misc_cb_t *cb);

void           usbdi_unbind_ack(     usbdi_misc_cb_t *cb,
                                     udi_status_t status);

typedef void   usbdi_unbind_ack_op_t( usbdi_misc_cb_t *cb,
                                     udi_status_t status);

/*
 * Opening USB Interfaces
 *
 * Opening an interface results in the allocation of bandwidth
 * for all pipes in the interface.
 *
 * The "alternate_intfc" field shall be set appropriately.  If the
 * interface being opened is the default interface, alternate_intfc
 * shall be 0.  If an alternate interface is being opened, alternate_intfc
 * shall be set to the alternate interface number as found in the
 * struct usb_interface_descriptor for the interface being selected.
 */

void              usbdi_intfc_open_req(usbdi_misc_cb_t *cb,
                                     udi_ubit8_t alternate_intfc,
                                     udi_ubit8_t open_flag );

typedef void usbdi_intfc_open_req_op_t(usbdi_misc_cb_t *cb);

void              usbdi_intfc_open_ack(usbdi_misc_cb_t *cb,
                                     udi_index_t n_edpt,
                                     udi_status_t status);

typedef void usbdi_intfc_open_ack_op_t(usbdi_misc_cb_t *cb,
                                     udi_index_t n_edpt,
                                     udi_status_t status);

/*
 * Closing USB Interfaces
 *
 * Closing a USB interface results in all bandwidth for the interface
 * being released and made available to other USB interfaces.
 *
 * The same CB is used for both opening and closing interfaces.
 * See "Opening USB Interfaces" above.
 */
void              usbdi_intfc_close_req(usbdi_misc_cb_t *cb);

typedef void usbdi_intfc_close_req_op_t(usbdi_misc_cb_t *cb);

void              usbdi_intfc_close_ack(usbdi_misc_cb_t *cb,
                                       udi_status_t status);

typedef void usbdi_intfc_close_ack_op_t(usbdi_misc_cb_t *cb,
                                       udi_status_t status);
```

```
/*
 * USB interrupt and bulk transfer requests
 */
typedef struct {
    udi_cb_t gcb;      /* UDI general control block */

    /*
     * data_buf points to the associated buffer where the data
     * will be transferred to/from.  The LDD sets the data_buf->buf_size
     * to the maximum number of bytes which may be transferred.  To
     * specify a zero length data transfer the it shall be set
     * by the LDD to zero.
     *
     * On completion the USBD sets the data_buf->buf_size field with the
     * actual number of bytes transferred.
     */
    udi_buf_t *data_buf;

    /*
     * Request timeout value in milliseconds.  Request timeout periods
     * begin when the USBD queues the request to the host controller
     * and NOT when the host controller issues the request to/from the
     * device.  A timeout value of zero specifies an infinite timeout
     * period.
     */
    udi_ubit32_t timeout;

    /*
     * Flags specific to this request.  The only valid flag for interrupt
     * and bulk requests is USBDI_XFER_SHORT_OK which specifies a final
     * transfer count which is less than the maximum number set in data_len
     * by the LDD will NOT generate an error and will NOT stall the pipe.
     */
    udi_ubit8_t xfer_flags;
#define USBDI_XFER_SHORT_OK   (1<<0)
#define USBDI_XFER_IN         (1<<1) /* transfer data from the device */
#define USBDI_XFER_OUT        (1<<2) /* transfer data to the device */

} usbdi_intr_bulk_xfer_cb_t;

#define USBDI_INTR_BULK_XFER_CB_NUM          2


void           usbdi_intr_bulk_xfer_req(     usbdi_intr_bulk_xfer_cb_t *cb);

typedef void   usbdi_intr_bulk_xfer_req_op_t(       usbdi_intr_bulk_xfer_cb_t *cb);

void           usbdi_intr_bulk_xfer_ack(     usbdi_intr_bulk_xfer_cb_t *cb);

typedef void   usbdi_intr_bulk_xfer_ack_op_t(       usbdi_intr_bulk_xfer_cb_t *cb);

usbdi_intr_bulk_xfer_ack_op_t                usbdi_intr_bulk_xfer_ack_unused;

void           usbdi_intr_bulk_xfer_nak(     usbdi_intr_bulk_xfer_cb_t *cb,
                                             udi_status_t status);

typedef void   usbdi_intr_bulk_xfer_nak_op_t(       usbdi_intr_bulk_xfer_cb_t *cb,
                                             udi_status_t status);

usbdi_intr_bulk_xfer_nak_op_t                usbdi_intr_bulk_xfer_nak_unused;

/*
 * USB control transfer requests
 *
 * The device's default pipe is accessed by sending
 * usbdi_control_xfer_cb's to the interface channel.
 */
typedef struct {
    udi_cb_t gcb;      /* UDI general control block */
    /*
     * The "request" union is provided to support control endpoints
     * other than the default which may not use the usb_device_request_t
     * request format.  This is the data which will be sent to the device
     * during the set up phase of the control request.
     */
    union {
       usb_device_request_t device_request; /* Used by the default endpoint */
       udi_ubit8_t request[8];                   /* Used by control endpoints other
                                            * than the default
                                            */
    } request;

    /*
     * data_buf and data_len
     *
     * data_buf points to the associated buffer to be used during the
     * data phase transfer (if any).  The LDD sets the data_len field
     * to the maximum number of bytes which may be transferred during the
     * data phase.  To specify a zero length data transfer the data_len
     * shall be set by the LDD to zero.  If the data_len is non-zero, the
     * data_buf shall contain a valid address.
     *
     * On completion the USBD sets the data_len field with the actual
     * number of bytes transferred.
     */
    udi_buf_t *data_buf;

    /*
     * Request timeout value in milliseconds.  Request timeout periods
     * begin when the USBD queues the request to the host controller
     * and NOT when the host controller issues the request to/from the
     * device.  A timeout value of zero specifies an infinite timeout
     * period.
     */
    udi_ubit32_t timeout;

    /*
     * Flags specific to this control request.  The LDD shall set
     * the direction of the transfer during the data phase (if any).
     * USBDI_XFER_SHORT_OK (see usbdi_intr_bulk_xfer_cb_t) is also
     * a valid flag.
     */
    udi_ubit8_t xfer_flags;

} usbdi_control_xfer_cb_t;

#define USBDI_CONTROL_XFER_CB_NUM     3


void           usbdi_control_xfer_req(       usbdi_control_xfer_cb_t *cb);

typedef void   usbdi_control_xfer_req_op_t(  usbdi_control_xfer_cb_t *cb);

void           usbdi_control_xfer_ack(       usbdi_control_xfer_cb_t *cb);

typedef void   usbdi_control_xfer_ack_op_t(  usbdi_control_xfer_cb_t *cb,
                                             udi_status_t status);

usbdi_control_xfer_ack_op_t                  usbdi_control_xfer_ack_unused;

void           usbdi_control_xfer_nak(       usbdi_control_xfer_cb_t *cb,
                                             udi_status_t status);

typedef void   usbdi_control_xfer_nak_op_t(  usbdi_control_xfer_cb_t *cb,
                                             udi_status_t status);

/*
 * USB isoc transfer requests
 *
 * usbdi_isoc_frame_request is a per-frame request struct.  An array
 * of usbdi_isoc_frame_request_t structs will be allocated automatically
 * when the usbdi_isoc_xfer_cb_t is allocated.  The number of
 * usbdi_isoc_request_t structs contained by this array is specified by
 * the LDD at driver initialization time when usbdi_isoc_xfer_cb_init()
 * is called.
 */
typedef struct {
    udi_ubit32_t frame_len;   /* Set by the LDD to the number of bytes to
                               * transfer in the frame.  Set by the USBD on
                               * completion with the actual number of bytes
                               * transferred in the frame.
                               */
    udi_status_t frame_status; /* Per frame status set by the USBD */
} usbdi_isoc_frame_request_t;

typedef struct {
    udi_cb_t gcb;      /* UDI general control block */

    /*
     * data_buf shall be set by the LDD to a valid buffer where data
     * will be transferred to/from.  This buffer shall be large enough
     * to store all data which will be transferred as described by the
     * frame_array.
     */
    udi_buf_t *data_buf;

    /*
     * The frame_array will be allocated automatically when the
     * usbdi_isoc_xfer_cb_t is allocated.  The frame_len fields for
     * each valid element in the array shall be set by the LDD.  It
     * is legal to specify a zero frame_len value.
     */
    usbdi_isoc_frame_request_t *frame_array;

    /*
     * frame_count is the number of valid entries in the frame_array.
     * This field is set by the LDD and may not exceed the maximum number
     * of entries in the array (see usbdi_isoc_xfer_cb_init()).
     */
    udi_ubit8_t frame_count;

    /*
     * Flags specific to this request.  The USBDI_XFER_ASAP flag is
     * used to specify that the request may be started with the next
     * available frame.  If the flag is set, the frame_number field will
     * be ignored by the USBD.  In addition to the USBDI_XFER_ASAP flag
     * the USBDI_XFER_SHORT_OK is also valid (see usbdi_intr_bulk_xfer_cb_t).
     */
    udi_ubit8_t xfer_flags;
#define USBDI_XFER_ASAP (1<<3)

    /*
     * Request timeout value in milliseconds.  Request timeout periods
     * begin when the USBD queues the request to the host controller
     * and NOT when the host controller issues the request on the USB.
     * A timeout value of zero specifies an infinite timeout period.
     */
    udi_ubit32_t timeout;

    /*
     * frame_number is set by the LDD to the frame number which this
     * request may begin with (see usbdi_frame_number_cb_t).  It is
     * set by the USBD at request completion with the frame number at
     * completion time.  This frame number is provided to allow LDDs to
     * synchronize requests queued to different isoc pipes.
     */
    udi_ubit32_t frame_number;

} usbdi_isoc_xfer_cb_t;

#define USBDI_ISOC_XFER_CB_NUM       4
```

```
void           usbdi_isoc_xfer_req(          udi_channel_t pipe_channel,
                                             usbdi_isoc_xfer_cb_t *cb);

typedef void   usbdi_isoc_xfer_req_op_t(     usbdi_isoc_xfer_cb_t *cb);

void           usbdi_isoc_xfer_ack(          usbdi_isoc_xfer_cb_t *cb);

typedef void   usbdi_isoc_xfer_ack_op_t(     usbdi_isoc_xfer_cb_t *cb);

usbdi_isoc_xfer_ack_op_t                     usbdi_isoc_xfer_ack_unused;

void           usbdi_isoc_xfer_nak(          usbdi_isoc_xfer_cb_t *cb,
                                             udi_status_t status);

typedef void   usbdi_isoc_xfer_nak_op_t(     usbdi_isoc_xfer_cb_t *cb,
                                             udi_status_t status);

usbdi_isoc_xfer_nak_op_t                     usbdi_isoc_xfer_nak_unused;

/*
 * Determine the current frame number.  This CB is provided to
 * allow isoc LDDs a means to determine the current frame number.
 * The frame number returned may not be the actual frame number
 * from the host controller.  It is a pseudo-frame number provided
 * by the USBD which will allow isoc LDDs to synchronize pipes which
 * may be on the same USB or may be on different USBs.
 */

void           usbdi_frame_number_req(       usbdi_misc_cb_t *cb);

typedef void   usbdi_frame_number_req_op_t(  usbdi_misc_cb_t *cb);

void           usbdi_frame_number_ack(       usbdi_misc_cb_t *cb,
                                             udi_ubit32_t frame_number);

typedef void   usbdi_frame_number_ack_op_t(  usbdi_misc_cb_t *cb,
                                             udi_ubit32_t frame_number);

usbdi_frame_number_ack_op_t                  usbdi_frame_number_ack_unused;


/*
 * Isochronous control block initialization (utility function):
 *
 * Use this routine to distribute the total length of data in the control
 * block's data buffer in the control block among the frame array length
 * in the control block
 *
 * To use this routine, both the data_buf and the frame_count fields of
 * the control block must be correct.
 */
udi_status_t   usbdi_frame_distrib(          usbdi_isoc_xfer_cb_t *cb);



/*
 * Determine the speed of the device.  This CB is provided to
 * allow LDDs a means to determine the speed of their associated device.
 * Valid speeds are: USBDI_DEVICE_SPEED_LOW, USBDI_DEVICE_SPEED_FULL,
 * and USBDI_DEVICE_SPEED_HIGH.
 */

#define USBDI_DEVICE_SPEED_LOW       0x01
#define USBDI_DEVICE_SPEED_FULL      0x02
#define USBDI_DEVICE_SPEED_HIGH      0x03

void           usbdi_device_speed_req(       usbdi_misc_cb_t *cb);

typedef void   usbdi_device_speed_req_op_t(  usbdi_misc_cb_t *cb);
```

```
void           usbdi_device_speed_ack(       usbdi_misc_cb_t *cb,
                                             udi_ubit8_t device_speed);

typedef void   usbdi_device_speed_ack_op_t(  usbdi_misc_cb_t *cb,
                                             udi_ubit32_t device_speed);

usbdi_device_speed_ack_op_t                  usbdi_device_speed_ack_unused;

/*
 * Reset the device associated with the interface_channel.  This
 * CB is provided to allow LDDs a means to recover from device hang
 * conditions.  THIS CB SHOULD NOT BE USED LIGHTLY!  The result will
 * be the device being re-enumerated.
 */

void           usbdi_reset_device_req(       usbdi_misc_cb_t *cb);

typedef void   usbdi_reset_device_req_op_t(  usbdi_misc_cb_t *cb);

void           usbdi_reset_device_ack(       usbdi_misc_cb_t *cb);

typedef void   usbdi_reset_device_ack_op_t(  usbdi_misc_cb_t *cb);

usbdi_reset_device_ack_op_t                  usbdi_reset_device_ack_unused;

/*
 * Abort a pipe
 *
 * All requests queued to the pipe will be returned to the LDD with
 * a status of UDI_STAT_ABORTED.  Only after all requests have been
 * aborted will the LDD's usbdi_pipe_abort_ack_op_t be called.
 * The pipe's state will be USBDI_STATE_IDLE on completion of the
 * abort process.
 */

void           usbdi_pipe_abort_req(         udi_channel_t pipe_channel,
                                             usbdi_misc_cb_t *cb);

typedef void   usbdi_pipe_abort_req_op_t(    usbdi_misc_cb_t *cb);

void           usbdi_pipe_abort_ack(         udi_channel_t pipe_channel,
                                             usbdi_misc_cb_t *cb);

typedef void   usbdi_pipe_abort_ack_op_t(    usbdi_misc_cb_t *cb);

usbdi_pipe_abort_ack_op_t                    usbdi_pipe_abort_ack_unused;


/*
 * Abort an interface
 *
 * All requests queued to all pipes of the interface will be returned
 * to the LDD with a status of UDI_STAT_ABORTED.  Only after all requests
 * have been aborted will the LDD's usbdi_intfc_abort_ack_op_t be called.
 * The interface's state will be USBDI_STATE_IDLE or USBDI_STATE_STALLED
 * on completion of the abort process.
 */

void           usbdi_intfc_abort_req(        udi_channel_t intfc_channel,
                                             usbdi_misc_cb_t *cb);

typedef void   usbdi_intfc_abort_req_op_t(   usbdi_misc_cb_t *cb);

void           usbdi_intfc_abort_ack(        usbdi_misc_cb_t *cb);

typedef void   usbdi_intfc_abort_ack_op_t(   usbdi_misc_cb_t *cb);

usbdi_intfc_abort_ack_op_t                   usbdi_intfc_abort_ack_unused;

/*
```

```
 * Getting and Setting States
 *
 * Devices, interfaces, pipes, and endpoints, all have a state associated
 * with them.  The following control block is used for setting and
 * getting states.
 */
typedef struct {
       udi_cb_t gcb;
       udi_ubit8_t state;
#define USBDI_STATE_ACTIVE           1
#define USBDI_STATE_STALLED          2
#define USBDI_STATE_IDLE             3
#define USBDI_STATE_HALTED           4
#define USBDI_STATE_CONFIGURED       (1 << 1)
#define USBDI_STATE_SUSPENDED        (1 << 2)
} usbdi_state_cb_t;

#define USBDI_STATE_CB_NUM           4


/*
 * Pipe get and set state
 *
 * Pipes support three states:
 *
 *    USBDI_STATE_ACTIVE - The pipe will accept transfer requests and will
 *              send requests to the device according to the type of
 *              the endpoint.
 *    USBDI_STATE_STALLED - The pipe will accept transfer requests but
 *              will not send requests to the device.
 *    USBDI_STATE_IDLE - The pipe will not accept transfer requests and will
 *              not send requests to the device.
 */

void           usbdi_pipe_state_set_req(     usbdi_state_cb_t *cb);

typedef void   usbdi_pipe_state_set_req_op_t(usbdi_state_cb_t *cb);

void           usbdi_pipe_state_set_ack(     usbdi_state_cb_t *cb,
                                             udi_status_t status);

typedef void   usbdi_pipe_state_set_ack_op_t(usbdi_state_cb_t *cb,
                                             udi_status_t status);

void           usbdi_pipe_state_get_req(     usbdi_state_cb_t *cb);

typedef void   usbdi_pipe_state_get_req_op_t(usbdi_state_cb_t *cb);

void           usbdi_pipe_state_get_ack(     usbdi_state_cb_t *cb);

typedef void   usbdi_pipe_state_get_ack_op_t(usbdi_state_cb_t *cb);

usbdi_pipe_state_get_ack_op_t                usbdi_pipe_state_get_ack_unused;

/*
 * Endpoint set and get state
 *
 * Endpoints support two states
 *
 *    USBDI_STATE_ACTIVE  - The endpoint is accepting requests.  Setting
 *              the endpoint's state to active will reset the data toggle
 *              and will result in a clear feature "endpoint_halt" device
 *              request being issued to default pipe.
 *    USBDI_STATE_HALTED -  The endpoint has been halted.  See the USB
 *              core spec for a description of halted and possible causes.
 *              This state is read-only.
 */

void           usbdi_edpt_state_set_req(     usbdi_state_cb_t *cb);

typedef void   usbdi_edpt_state_set_req_op_t(usbdi_state_cb_t *cb);
```

```
void           usbdi_edpt_state_set_ack(     usbdi_state_cb_t *cb,
                                             udi_status_t status);

typedef void   usbdi_edpt_state_set_ack_op_t(usbdi_state_cb_t *cb,
                                             udi_status_t status);

void           usbdi_edpt_state_get_req(     usbdi_state_cb_t *cb);

typedef void   usbdi_edpt_state_get_req_op_t(usbdi_state_cb_t *cb);

void           usbdi_edpt_state_get_ack(     usbdi_state_cb_t *cb);

typedef void   usbdi_edpt_state_get_ack_op_t(usbdi_state_cb_t *cb);

usbdi_edpt_state_get_ack_op_t                usbdi_edpt_state_get_ack_unused;

/*
 * Interface Set/Get State
 *
 * Setting the interface state results in all pipes of the interface
 * being moved to that state.  Once an interface has been moved to the
 * stalled or idle state, usbdi_pipe_state_set_req() may no longer be
 * used to change the state of the individual pipes.  Only when the
 * interface is in the active state may the pipe's state be changed with
 * usbdi_pipe_state_set_req().
 *
 * The state of the interface may be active even if one of more
 * of its pipes are stalled or idle.
 */

void           usbdi_intfc_state_set_req(    usbdi_state_cb_t *cb);

typedef void   usbdi_intfc_state_set_req_op_t(usbdi_state_cb_t *cb);

void           usbdi_intfc_state_set_ack(    usbdi_state_cb_t *cb,
                                             udi_status_t status);

typedef void   usbdi_intfc_state_set_ack_op_t(usbdi_state_cb_t *cb,
                                             udi_status_t status);

usbdi_intfc_state_set_ack_op_t                       usbdi_intfc_state_set_ack_unused;

void           usbdi_intfc_state_get_req(    usbdi_state_cb_t *cb);

typedef void   usbdi_intfc_state_get_req_op_t(usbdi_state_cb_t *cb);

void           usbdi_intfc_state_get_ack(    usbdi_state_cb_t *cb,
                                             udi_status_t status);

typedef void   usbdi_intfc_state_get_ack_op_t(usbdi_state_cb_t *cb,
                                             udi_status_t status);

usbdi_intfc_state_get_ack_op_t                       usbdi_intfc_state_get_ack_unused;

/*
 * Getting Device States
 *
 * At times an LDD may find it necessary to determine what state
 * its associated USB device is in.  Supported device states are
 * USBDI_STATE_CONFIGURED and USBDI_STATE_SUSPENDED.
 */

void           usbdi_device_state_get_req(          usbdi_state_cb_t *cb);

typedef void   usbdi_device_state_get_req_op_t(     usbdi_state_cb_t *cb);

void           usbdi_device_state_get_ack(          usbdi_state_cb_t *cb);

typedef void   usbdi_device_state_get_ack_op_t(     usbdi_state_cb_t *cb);
```

```
usbdi_device_state_get_ack_op_t                     usbdi_device_state_get_ack_unused;

/*
 * Retrieving Descriptors
 *
 * Any descriptor may be retrieved with the usbdi_desc_req() mechanisms.
 *
 * The USBD manages the retrieval of these descriptors in an
 * implementation-specific way.  Some USBD implementations may choose
 * to cache the configuration descriptors while others may choose to
 * read the descriptor from the device as needed.
 *
 * The USBD allocates the needed memory to create a copy of the
 * requested descriptor (as a movable memory block).  It is the
 * responsibility of the LDD to release this memory by calling
 * udi_mem_free();
 *
 * This mechanism allows for the retrieval of individual descriptors
 * which are returned by the device as part of the configuration
 * descriptor, such as the interface descriptor.
 *
 * The "desc_type" may be device, configuration, string, interface,
 * endpoint, USB class specific, or vendor specific.
 *
 * The "desc_index" field of the usbdi_desc_cb_t is zero based.
 * When retrieving descriptors contained in the configuration that
 * are part of an interface the "desc_index" refers to the instance
 * of the descriptor for the interface related to the channel.  For
 * example, to retrieve the first endpoint descriptor of an interface
 * the "desc_index" may be zero even if the interface is not the
 * first interface of the configuration.
 *
 * The USB configuration descriptor alone may be retrieved or all
 * of the configuration descriptors may be retrieved by setting the
 * "desc_length" appropriately.
 *
 * Refer to the USB core specification for more details on desc_type,
 * desc_index, and languageID (desc_ID field).
 */
typedef struct {
       udi_cb_t gcb;
       udi_ubit8_t desc_type;
#define USB_DESC_TYPE_DEVICE         0x01
#define USB_DESC_TYPE_CONFIG         0x02
#define USB_DESC_TYPE_STRING         0x03
#define USB_DESC_TYPE_INTFC          0x04
#define USB_DESC_TYPE_EDPT           0x05
       udi_ubit8_t desc_index;
       udi_ubit16_t desc_ID;  /* Language ID for string descriptors,
                               * functional device ID for all other
                               * descriptors.
                               */
       udi_buf_t desc_buf;    /* Returned descriptor */
       udi_ubit16_t desc_length;
} usbdi_desc_cb_t;

#define USBDI_DESC_CB_NUM            5


void           usbdi_desc_req(       usbdi_desc_cb_t *cb);

typedef void   usbdi_desc_req_op_t(  usbdi_desc_cb_t *cb);

void           usbdi_desc_ack(       usbdi_desc_cb_t *cb,
                                     udi_status_t status);

typedef void   usbdi_desc_ack_op_t(  usbdi_desc_cb_t *cb,
                                     udi_status_t status);

usbdi_desc_ack_op_t                  usbdi_desc_ack_unused;
```

```
/*
 * Change the device's configuration setting
 *
 * All USB interfaces contained by the current configuration shall
 * be in the closed state before usbdi_config_set_req() may be
 * performed.
 *
 * Once the configuration has been changed all LDD's bound to the
 * previous configuration will be unbound and the new configuration will
 * be rebound.
 *
 * A LDD may determine the current configuration value by issuing
 * a USB_DEVICE_REQUEST_GET_CONFIGURATION request.
 */

void    usbdi_config_set_req(                usbdi_misc_cb_t *cb,
                                             udi_ubit16_t config_value);

typedef void   usbdi_config_set_req_op_t(    usbdi_misc_cb_t *cb);

void           usbdi_config_set_ack(         usbdi_misc_cb_t *cb,
                                             udi_status_t status);

typedef void   usbdi_config_set_ack_op_t(    usbdi_misc_cb_t *cb,
                                             udi_status_t status);

usbdi_config_set_ack_op_t                    usbdi_config_set_ack_unused;

/*
 * Asynchronous Events
 *
 * Asynchronous event notification is provided to notify LDDs of
 * events they may not have initiated.  These events include device
 * suspended, device wakeup, and USB bandwidth needed.
 *
 * All LDDs that have opened interfaces with the USBDI_INTFC_OPEN_ASYNC_EVENT
 * flag set will be notified when the device has been suspended or woken-up.
 * All LDDs with alternate interface's will be notified when USB bandwidth
 * is needed.  A well-behaved LDD that can reduce its USB bandwidth
 * consumption by switching to an alternate interface may do so when
 * this asynchronous event is received and before calling
 * usbdi_async_event_res().
 */

/*
 * async_event may be one of the following:
 */
#define USBDI_ASYNC_SUSPEND          1
#define USBDI_ASYNC_WAKEUP           2
#define USBDI_ASYNC_BANDWIDTH_NEEDED  3

void           usbdi_async_event_ind(        udi_channel_t intfc_channel,
                                             usbdi_misc_cb_t *cb,
                                             udi_ubit16_t async_event);

typedef void   usbdi_async_event_ind_op_t(   usbdi_misc_cb_t *cb,
                                             udi_ubit16_t async_event);

usbdi_async_event_ind_op_t                   usbdi_async_event_ind_unused;

void           usbdi_async_event_res(        usbdi_misc_cb_t *cb);

typedef void   usbdi_async_event_res_op_t(   usbdi_misc_cb_t *cb);


/*
 * Channel operations for LDD side of bind channels and other USB
 * interface channels.
 */
typedef struct {
    udi_channel_event_ind_op_t       *udi_channel_event_ind_op;
    usbdi_bind_ack_op_t               *usbdi_bind_ack_op;
    usbdi_unbind_ack_op_t             *usbdi_unbind_ack_op;
    usbdi_intfc_open_ack_op_t         *usbdi_intfc_open_ack_op;
    usbdi_intfc_close_ack_op_t        *usbdi_intfc_close_ack_op;
    usbdi_frame_number_ack_op_t       *usbdi_frame_number_ack_op;
    usbdi_device_speed_ack_op_t      *usbdi_device_speed_ack_op;
    usbdi_reset_device_ack_op_t       *usbdi_reset_device_ack_op;
    usbdi_intfc_abort_ack_op_t        *usbdi_intfc_abort_ack_op;
    usbdi_intfc_state_set_ack_op_t    *usbdi_intfc_state_set_ack_op;
    usbdi_intfc_state_get_ack_op_t    *usbdi_intfc_state_get_ack_op;
    usbdi_desc_ack_op_t               *usbdi_desc_ack_op;
    usbdi_device_state_get_ack_op_t   *usbdi_device_state_get_ack_op;
    usbdi_config_set_ack_op_t         *usbdi_config_set_ack_op;
    usbdi_async_event_ind_op_t        *usbdi_async_event_ind_op;
} usbdi_ldd_intfc_ops_t;

#define USBDI_LDD_INTFC_OPS_NUM       1


/*
 * Interface operations for LDD side of pipe channels.
 */
typedef struct {
       udi_channel_event_ind_op_t           *udi_channel_event_ind_op;
       usbdi_intr_bulk_xfer_ack_op_t        *usbdi_intr_bulk_xfer_ack_op;
       usbdi_intr_bulk_xfer_nak_op_t        *usbdi_intr_bulk_xfer_nak_op;
       usbdi_control_xfer_ack_op_t          *usbdi_control_xfer_ack_op;
       usbdi_isoc_xfer_ack_op_t             *usbdi_isoc_xfer_ack_op;
       usbdi_isoc_xfer_nak_op_t             *usbdi_isoc_xfer_nak_op;
       usbdi_pipe_abort_ack_op_t            *usbdi_pipe_abort_ack_op;
       usbdi_pipe_state_set_ack_op_t        *usbdi_pipe_state_set_ack_op;
       usbdi_pipe_state_get_ack_op_t        *usbdi_pipe_state_get_ack_op;
       usbdi_edpt_state_set_ack_op_t        *usbdi_edpt_state_set_ack_op;
       usbdi_edpt_state_get_ack_op_t        *usbdi_edpt_state_get_ack_op;
} usbdi_ldd_pipe_ops_t;

#define USBDI_LDD_PIPE_OPS_NUM        2


/*
 * Interface operations for USBD side of bind channels and other USB
 * interface channels.
 */
typedef struct {
       udi_channel_event_ind_op_t           *udi_channel_event_ind_op;
       usbdi_bind_req_op_t                  *usbdi_bind_req_op;
       usbdi_unbind_req_op_t                *usbdi_unbind_req_op;
       usbdi_intfc_open_req_op_t            *usbdi_intfc_open_req_op;
       usbdi_intfc_close_req_op_t           *usbdi_intfc_close_req_op;
       usbdi_frame_number_req_op_t          *usbdi_frame_number_req_op;
       usbdi_reset_device_req_op_t          *usbdi_reset_device_req_op;
       usbdi_intfc_abort_req_op_t           *usbdi_intfc_abort_req_op;
       usbdi_intfc_state_set_req_op_t       *usbdi_intfc_state_set_req_op;
       usbdi_intfc_state_get_req_op_t       *usbdi_intfc_state_get_req_op;
       usbdi_desc_req_op_t                  *usbdi_desc_req_op;
       usbdi_device_state_get_req_op_t      *usbdi_device_state_get_req_op;
       usbdi_config_set_req_op_t            *usbdi_config_set_req_op;
       usbdi_async_event_res_op_t           *usbdi_async_event_res_op;
} usbdi_usbd_intfc_ops_t;


/*
 * Interface operations for USBD side of pipe channels.
 */
typedef struct {
       udi_channel_event_ind_op_t           *udi_channel_event_ind_op;
       usbdi_intr_bulk_xfer_req_op_t        *usbdi_intr_bulk_xfer_req_op;
       usbdi_control_xfer_req_op_t          *usbdi_control_xfer_req_op;
       usbdi_isoc_xfer_req_op_t             *usbdi_isoc_xfer_req_op;
       usbdi_pipe_abort_req_op_t            *usbdi_pipe_abort_req_op;
       usbdi_pipe_state_set_req_op_t        *usbdi_pipe_state_set_req_op;
```

```
        usbdi_pipe_state_get_req_op_t         *usbdi_pipe_state_get_req_op;
        usbdi_edpt_state_set_req_op_t         *usbdi_edpt_state_set_req_op;
        usbdi_edpt_state_get_req_op_t         *usbdi_edpt_state_get_req_op;
} usbdi_usbd_pipe_ops_t;

#endif /* OPEN_USBDI_VERSION */
#endif /* _OPEN_USBDI_H_ */
```

8 Appendix B: Sample Driver (usbdi_printer.c)

```
#define OPEN_USBDI_VERSION 0x09b
#define UDI_VERSION 0x095
#include <udi.h>
#include <open_usbdi.h>
#include "usbdi_printer.h"

#ifndef UDI_NULL
#define UDI_NULL 0
#endif

/*
 * This driver is preliminary!  It compiles but
 * certainly doesn't work.  It is being distributed
 * for review purposes only and may not be used as
 * as a basis for any driver development.
 *
 * This file contains a uni-directional printer driver for USB
 * devices that conform to the "Universal Serial Bus Device Class
 * Definition for Printer Devices", ver 1.0.
 *
 * This Logical-Device Driver (LDD) conforms to OpenUSBDI (ver 0.9b)
 * and UDI (ver 1.0) including the UDI Generic I/O Metalanguage.
 *
 * UDI does not (currently) provide a printer metalanguage, therefore
 * this driver is using the UDI Generic I/O Metalanguage (GIO).
 *
 * It is expected that most third party USB device vendors who
 * write drivers for vendor unique devices will use the GIO interface.
 */
/*
 * Operation structures
 *
 * We declare a few of these statically, then tie them together in a
 * static array later on.  This static array will then be tied into the
 * udi_init_info structure.
 */
const udi_gio_provider_ops_t print_gio_provider_ops = {
    print_gio_provider_channel_event_ind,
    print_gio_bind_req,
    print_gio_unbind_req,
    print_gio_xfer_req,
    udi_gio_event_res_unused
};

static usbdi_ldd_intfc_ops_t print_usbdi_ldd_intfc_ops = {
    print_usbdi_ldd_intfc_channel_event_ind,
    print_usbdi_bind_ack,
    print_usbdi_unbind_ack,
    print_usbdi_intfc_open_ack,
    print_usbdi_intfc_close_ack,
    usbdi_frame_number_ack_unused,
    usbdi_device_speed_ack_unused,
    usbdi_reset_device_ack_unused,
    print_usbdi_intfc_abort_ack,
    usbdi_intfc_state_set_ack_unused,
    usbdi_intfc_state_get_ack_unused,
    print_usbdi_desc_ack,
    usbdi_device_state_get_ack_unused,
    usbdi_config_set_ack_unused,
    print_usbdi_async_event_ind
};

static usbdi_ldd_pipe_ops_t print_usbdi_ldd_pipe_ops = {
    print_usbdi_ldd_pipe_channel_event_ind,
    print_usbdi_intr_bulk_xfer_ack,
    print_usbdi_intr_bulk_xfer_nak,
    print_usbdi_control_xfer_ack,
    usbdi_isoc_xfer_ack_unused,
    usbdi_isoc_xfer_nak_unused,
    usbdi_pipe_abort_ack_unused,
    print_usbdi_pipe_state_set_ack,
    print_usbdi_pipe_state_get_ack,
    print_usbdi_edpt_state_set_ack,
    print_usbdi_edpt_state_get_ack
};


/*
 * This array ties all of the operation initialization structures together
 */
static udi_ops_init_t print_ops_list[] = {
  {
    PRINT_USBDI_INTFC_OPS_IDX,
    PRINT_USBDI_META_IDX,
    USBDI_LDD_INTFC_OPS_NUM,
    0,
    (udi_ops_vector_t *)&print_usbdi_ldd_intfc_ops
  },
  {
    PRINT_USBDI_PIPE_OPS_IDX,
    PRINT_USBDI_META_IDX,
    USBDI_LDD_PIPE_OPS_NUM,
    0,
    (udi_ops_vector_t *)&print_usbdi_ldd_pipe_ops
  },
  {
    PRINT_GIO_PROVIDER_OPS_IDX,
    PRINT_GIO_META_IDX,
    UDI_GIO_PROVIDER_OPS_NUM,
    0,
    (udi_ops_vector_t *)&print_gio_provider_ops
  },
  {
    0
  }
};


/*
 * Control block structures
 *
 * We declare a bunch of these statically, then tie them together in a
 * static array later on.  This static array will then be tied into the
 * udi_init_info structure.
 */
static udi_cb_init_t print_cb_init_list[] = {
  {
    PRINT_USBDI_MISC_CB_IDX,
    PRINT_USBDI_META_IDX,
    USBDI_MISC_CB_NUM,
    PRINT_USBDI_MISC_CB_SCRATCH_SIZE,
    0,
    NULL
  },
  {
    PRINT_USBDI_BULK_XFER_CB_IDX,
    PRINT_USBDI_META_IDX,
    USBDI_INTR_BULK_XFER_CB_NUM,
    PRINT_USBDI_BULK_XFER_CB_SCRATCH_SIZE,
    0,
    NULL
  },
  {
    PRINT_USBDI_STATE_CB_IDX,
    PRINT_USBDI_META_IDX,
    USBDI_STATE_CB_NUM,
    PRINT_USBDI_STATE_CB_SCRATCH_SIZE,
    0,
    NULL
  },
  {
    PRINT_USBDI_CONTROL_XFER_CB_IDX,
    PRINT_USBDI_META_IDX,
    USBDI_CONTROL_XFER_CB_NUM,
    PRINT_USBDI_CONTROL_XFER_CB_SCRATCH_SIZE,
    0,
    NULL
  },
  {
    PRINT_GIO_BIND_CB_IDX,
    PRINT_GIO_META_IDX,
    UDI_GIO_BIND_CB_NUM,
    PRINT_GIO_BIND_CB_SCRATCH_SIZE,
    0,
    NULL
  },
  {
    PRINT_GIO_XFER_CB_IDX,
    PRINT_GIO_META_IDX,
    UDI_GIO_XFER_CB_NUM,
    PRINT_GIO_XFER_CB_SCRATCH_SIZE,
    0,
    NULL
  },
  {
    PRINT_GIO_EVENT_CB_IDX,
    PRINT_GIO_META_IDX,
    UDI_GIO_EVENT_CB_NUM,
    PRINT_GIO_EVENT_CB_SCRATCH_SIZE,
    0,
    NULL
  },
  {
    0
  }
};


/*
 * Management operations
 */
static udi_mgmt_ops_t print_mgmt_ops = {
    udi_static_usage,
    udi_enumerate_no_children,
    print_udi_devmgmt_req,
    print_udi_final_cleanup_req
};

/*
 * Primary region initializer
 */
static udi_primary_init_t print_primary_init = {
    &print_mgmt_ops,                 /* Mgmt ops vector */
    0,                               /* Mgmt CB scratch requirement */
    0,                               /* Enumeration attr list (unused) */
    PRINT_UDI_PRIMARY_REGION_SIZE,
    0
};


/*
 * The main initialization structure
 */
static udi_init_t init_info = {
    &print_primary_init,      /* Primary init info */
    NULL,                     /* Secondary init info */
    print_ops_list,           /* ops init list */
    print_cb_init_list,              /* CBinit list */
    NULL,                     /* GCB init list */
    NULL                      /* CB select list */
};

/*
 * Function: print_usbdi_intfc_channel_event_ind()
 *
 * This function is called by the UDI Management Agent (MA) when there are
 * channel events on this channel.  Channel events may include channel bindings,
 * unbindings, abort notification for channel ops, and constraint changes.
 */
void
print_usbdi_ldd_intfc_channel_event_ind( udi_channel_event_cb_t *cb ) {

    /*
     * The printer_context struct was allocated by UDI.  The
     * size had been given to the MA in the initialization data structs
     */
    struct printer_context *printer = UDI_GCB(cb)->context;

    switch (cb->event) {

    case (UDI_CHANNEL_BOUND):

        /* This is the first call we get when this channel is opened */
        printer->usbdi_channel_cb = cb;
        udi_cb_alloc(print_usbdi_bind_cb_alloc_call,
                     UDI_GCB(cb),
                     PRINT_USBDI_MISC_CB_IDX,
                     UDI_NULL);
        break;

    /*
     * None of these next events should ever occur from the USBD, but
     * we'll go ahead and recognize them here
     */
    case (UDI_CHANNEL_CLOSED):
        /*
         * If we were in the middle of a GIO xfer when this came in, cancel it,
         * based on the assumption that the request won't be completing.
         */
        if (printer->flags & PRINT_FLAGS_GIO_XFER)  {
            udi_gio_xfer_nak(printer->gio_xfer_cb, UDI_STAT_RESOURCE_UNAVAIL);
            printer->gio_xfer_cb = (udi_gio_xfer_cb_t *)UDI_NULL;
            printer->flags &= ~PRINT_FLAGS_GIO_XFER;
        }

        /* Fall through to the event completion call */

    case (UDI_CHANNEL_OP_ABORTED):
    case (UDI_CONSTRAINTS_CHANGED):
    default:
        udi_channel_event_complete(cb, UDI_OK);
        break;
    }
}


/*
 * Function: print_usbdi_bind_cb_alloc_call()
 *
 * Callback function given to udi_cb_alloc() in
 * print_usbdi_intfc_channel_event_ind().
 */
static void
print_usbdi_bind_cb_alloc_call(udi_cb_t *gcb, udi_cb_t *new_cb)
{
    usbdi_misc_cb_t *usbdi_bind_cb = UDI_MCB(new_cb, usbdi_misc_cb_t);

    /*
     * Request to complete the binding for the first interface.
     */
    usbdi_bind_req(usbdi_bind_cb);

    /*
     * Execution continues in print_usbdi_bind_ack().
     */
}


/*
 * Function:  print_usbdi_bind_ack()
 *
 * Interface op function for the usbdi_ldd_intfc_ops_t bind operation.
 * Called in response to a usbdi_bind_req() call.
 */
static void
print_usbdi_bind_ack(usbdi_misc_cb_t *cb,
                    udi_index_t n_intfc,
                    udi_status_t status)
{
    struct printer_context *printer = UDI_GCB(cb)->context;

    /*
     * Save the CB.  It will be used when the driver unbinds from USBD.
     * We'll also use the bind channel CB to cancel channel operations
     */
    printer->usbdi_bind_cb = cb;
    printer->usbdi_intfc_channel = UDI_GCB(cb)->channel;

    /*
     * n_intfc is the number of interfaces associated with this LDD.
     * In this case it had better be one.
     */
    if (n_intfc != PRINTER_INTFC_COUNT) {

        /*
         * If the number of interfaces doesn't match we need to fail
         * the binding.
         */
        udi_channel_event_complete(printer->usbdi_channel_cb,
                                   UDI_STAT_CANNOT_BIND);
        printer->usbdi_channel_cb = (udi_channel_event_cb_t *)UDI_NULL_CHANNEL;
        print_free_printer(printer);
        return;
    }

    /*
     * Read in the descriptors for this interface.
     */
    print_usbdi_read_descriptors(printer);
}


/*
 * OpenUSBDI Read Descriptor Example
 *
 * This driver reads the device and interface descriptor at driver
 * bind time.  Based on the descriptors the driver may determine that
 * the driver is not suited for the device and fail the binding.
 */

/*
 * Function: print_usbdi_read_descriptors()
 *
 * Read the device and interface descriptor.
 * This is done only once at driver bind time.
 */
static void
print_usbdi_read_descriptors(struct printer_context *printer)
{
    /*
     * Allocate a usbdi_desc_cb_t.  This CB will be used to read
     * both the device descriptor and the interface descriptor.
     */
    udi_cb_alloc(print_usbdi_desc_cb_alloc_call,
                 UDI_GCB(printer->usbdi_bind_cb),
                 PRINT_USBDI_DESC_CB_IDX,
                 printer->usbdi_intfc_channel);

    /*
     * Execution continues in print_usbdi_desc_cb_alloc_call().
     */
}

/*
 * Function:  print_usbdi_desc_cb_alloc_call()
 *
 * Callback function given to udi_cb_alloc() by
 * print_usbdi_read_descriptors().
 *
 * Set up the CB to read the device descriptor.
 */
static void
print_usbdi_desc_cb_alloc_call(udi_cb_t *gcb, udi_cb_t *new_cb)
{
    struct printer_context *printer = gcb->context;
    struct printer_usbdi_desc_cb_scratch *scratch = new_cb->scratch;
    usbdi_desc_cb_t *cb = UDI_MCB(new_cb, usbdi_desc_cb_t);

    /*
     * Save the buffer pointer for the device descriptor in
     * the CBs scratch space.
     */
    scratch->desc_buf = (udi_buf_t *)&printer->usb_device_descriptor;

    /*
     * Set up the CB to request the USB device descriptor.
     */
    cb->desc_type = USB_DESC_TYPE_DEVICE;
    cb->desc_index = 0;
    cb->desc_ID = 0;
    cb->desc_length = sizeof(struct usb_device_descriptor);
    usbdi_desc_req(cb);

    /*
     * Once the USBD has completed the operation it will
     * call print_usbdi_desc_ack() which was registered
     * in the usbdi_ldd_intfc_ops_t.
     *
     * print_usbdi_desc_ack will store the descriptor in the
     * print_context structure, then call
     * print_usbdi_read_interface_descriptor()
     */
}

/*
 * Function:  print_usbdi_read_interface_descriptor()
 *
 * Given a usbdi_desc_cb_t, perform a usbdi_desc_req()
 * requesting the USB interface descriptor.
 */
static void
print_usbdi_read_interface_descriptor(usbdi_desc_cb_t *cb)
{
    struct printer_context *printer = UDI_GCB(cb)->context;
    struct printer_usbdi_desc_cb_scratch *scratch = UDI_GCB(cb)->scratch;

    /*
     * Have we already read the default interface descriptor?
     */
    if (printer->usb_interface_descriptor ==
       (struct usb_interface_descriptor *)NULL) {
       /*
        * Set up the CB to read the default interface descriptor.
        */
       scratch->desc_buf = (udi_buf_t *)&printer->usb_interface_descriptor;
       cb->desc_type = USB_DESCRIPTOR_TYPE_INTERFACE;
       cb->desc_index = 0;
       cb->desc_ID = 0;
       cb->desc_length = sizeof(struct usb_interface_descriptor);
       usbdi_desc_req(cb);
       return;

       /*
        * Execution continues in print_usbdi_desc_ack() once the
        * USBD completes the read of the interface descriptor.
        *
        * print_usbdi_desc_ack() will call this routine again...
        * execution will continue below.
        */
    }

    /*
     * We are here once we have read the interface descriptor.
     *
     * This driver only supports the uni-directional printer
     * interface.
     *
     * Have we found the uni-directional interface?
     */
    if (printer->usb_interface_descriptor->bInterfaceProtocol !=
       INTERFACE_PROTOCOL_PRINTER_UNIDIRECTIONAL) {

       /*
        * Still haven't found the unidirectional interface.
        * Try the next one.
        *
        * Free the interface descriptor that was just read.
        */
       udi_mem_free(printer->usb_interface_descriptor);
       printer->usb_interface_descriptor =
           (struct usb_interface_descriptor *)NULL;

       /*
        * Move on to the next one.
        */
       cb->desc_index++;
       usbdi_desc_req(cb);
       return;

       /*
        * Execution continues in print_usbdi_desc_ack() once the
        * USBD completes the read of the interface descriptor.
        */
    }

    /*
     * If we make it here, we have successfully read the interface
     * descriptor for the uni-directional setting for the interface.
     * This is required before the binding of the driver may be completed.
     *
     * Release the usbdi_desc_cb.
     */
    udi_cb_free(UDI_GCB(cb));

    /*
     * The binding is complete.
     */
    udi_complete_channel_event(printer->usbdi_channel_cb, UDI_OK);
    printer->usbdi_channel_cb = (udi_channel_event_cb_t *)NULL;

    /*
     * At this point the interface has been bound.  The next step is
     * to open the pipes.  This happens when the interface is opened
     * via the GIO op print_gio_bind_req().
     */
}

/*
 * Function:  print_gio_provider_channel_event_ind()
 *
 * This is the channel event routine established in our GIO-specific
 * udi_ops_init_t data structure.
 *
 * This routine is called by the MA once the channel between this LDD's region
 * and the GIO region is established (UDI_CHANNEL_BOUND).
 *
 * It is also called when the GIO client aborts a channel operation
 * (UDI_CHANNEL_OP_ABORTED).
 *
 * It is also called as the MA closes the channel to the GIO region abruptly,
 * possibly because this LDD is being unloaded.
 */
static void
print_gio_provider_channel_event_ind(udi_channel_event_cb_t *cb) {

    struct printer_context *printer = UDI_GCB(cb)->context;

    switch( cb->event )  {

    /* The GIO client has aborted an operation */
    case( UDI_CHANNEL_OP_ABORTED ):
       udi_channel_op_abort(printer->usbdi_pipe_channel,
                            UDI_GCB(printer->bulk_xfer_cb));
       break;

    /*
     * If the GIO channel has been closed, no more operations will be permitted over
     * that channel.  Zero out that channel variable.
     */
    case( UDI_CHANNEL_CLOSED ):
       if (printer->gio_xfer_cb != (udi_gio_xfer_cb_t *) UDI_NULL) {
           udi_gio_xfer_nak(printer->gio_xfer_cb, UDI_STAT_RESOURCE_UNAVAIL);
       }

       printer->gio_xfer_cb = (udi_gio_xfer_cb_t *)UDI_NULL;
       printer->gio_bind_cb = (udi_gio_bind_cb_t *)UDI_NULL;
       break;

    /* For these events, do nothing special */
    case( UDI_CHANNEL_BOUND ):
    case( UDI_CONSTRAINTS_CHANGED ):
    default:
       break;

    }
    /*
     * In any case, go ahead and complete the event.
     */
    udi_channel_event_complete( cb, UDI_OK );

}


/*
 * Function: print_gio_bind_req()
 *
 * Given as the udi_gio_provider_ops_t bind_req function.  This
 * function is the equivalent to the printer driver's open function.
 */
static void
print_gio_bind_req(udi_gio_bind_cb_t *gio_bind_cb)
{
    struct printer_context *printer = UDI_GCB(gio_bind_cb)->context;
    /*
     * Make sure we can perform the binding:
     *
     * Make sure that the driver hasn't already been bound.
     * Only allow one user at a time.
     */
    if (printer->flags & PRINT_FLAGS_GIO_BOUND) {
       udi_gio_bind_ack(gio_bind_cb, 0, 0, UDI_STAT_BUSY);
       return;
    }

    /*
     * Execution continues in print_usbdi_intfc_open.
     */
    printer->gio_bind_cb = gio_bind_cb;
    printer->flags |= PRINT_FLAGS_GIO_BOUND;
    print_usbdi_intfc_open(UDI_GCB(gio_bind_cb));
}

/*
 * Function: print_gio_unbind_req()
 *
 * Given as the udi_gio_provider_ops_t unbind_req_function.  This
 * function is the equivalent of the printer driver's close function.
 */
static void
print_gio_unbind_req(udi_gio_bind_cb_t *gio_bind_cb)
{
    struct printer_context *printer = UDI_GCB(gio_bind_cb)->context;
    printer->gio_bind_cb = gio_bind_cb;

    /*
     * Do we need to handle the case where a close is requested
     * before the open completes?
     */

    if (!(printer->flags & PRINT_FLAGS_GIO_XFER)) {
       /*
        * We make it here if there are no outstanding requests
        * and we are ready to close the interface.
        */
       printer->flags &= ~PRINT_FLAGS_GIO_BOUND;
       print_usbdi_intfc_close(UDI_GCB(gio_bind_cb));
       udi_gio_unbind_ack(gio_bind_cb);

    }
    else {
       /*
        * There is an outstanding request.  Attempt to cancel it.
        */
       udi_channel_op_abort(printer->usbdi_pipe_channel,
                            UDI_GCB(printer->bulk_xfer_cb));
       printer->flags &= ~PRINT_FLAGS_GIO_BOUND;
    }
}

/*
 * Function Name:  print_gio_xfer_req()
 *
 * Function Description:
 *    This function is the GIO xfer req provider function that was
 *    registered for this LDD in the udi_gio_provider_ops_t.  It is
 *    called in response to the GIO client writing data to the printer.
 *    Only write operations are allowed.
 */
static void
print_gio_xfer_req(udi_gio_xfer_cb_t *gio_cb)
{
    struct printer_context *printer = UDI_GCB(gio_cb)->context;
    usbdi_intr_bulk_xfer_cb_t *bulk_cb = printer->bulk_xfer_cb;
    struct printer_pipe_context *pipe = UDI_GCB(bulk_cb)->context;

    /*
     * Make sure this is a write operation.
     */
    if (!(gio_cb->op & UDI_GIO_DIR_WRITE)) {
       udi_gio_xfer_nak(gio_cb, UDI_STAT_NOT_SUPPORTED);
       return;
    }

    /*
     * Make sure that we aren't already performing a transfer.
     */
    if (printer->flags & PRINT_FLAGS_GIO_XFER) {
       udi_gio_xfer_nak(gio_cb, UDI_STAT_BUSY);
       return;
    }

    /*
     * Save away the gio_xfer_cb pointer.
     */
    printer->flags |= PRINT_FLAGS_GIO_XFER;
    printer->gio_xfer_cb = gio_cb;

    bulk_cb->data_buf = gio_cb->data_buf;

    /*
     * Send it on down.
     */
    usbdi_intr_bulk_xfer(bulk_cb);

    /*
     * Execution continues in print_usbdi_intr_bulk_xfer_ack()
     */
}


/*
 * Function:  print_usbdi_unbind_ack()
 *
 * Interface op function for the usbdi_ldd_intfc_ops_t unbind operation.
 * Called in response to a usbdi_unbind_req() call.
 */
static void
print_usbdi_unbind_ack(usbdi_misc_cb_t *cb, udi_status_t status)
{
    struct printer_context *printer = UDI_GCB(cb)->context;

    /*
     * Save the CB.  It will be used when the driver unbinds from USBD.
     */
    printer->usbdi_bind_cb = cb;
}

/*
 * Function: print_usbdi_intfc_close_ack()
 *
 * Called in response to the driver calling usbdi_intfc_close_req().
 */
static void
print_usbdi_intfc_close_ack(usbdi_misc_cb_t *cb,
                          udi_status_t status)
{
    struct printer_context *printer = UDI_GCB(cb)->context;
    struct printer_usbdi_intfc_cb_scratch *scratch = UDI_GCB(cb)->scratch;
    int i;

    /*
     * Update our usbdi_intfc_cb pointer.  It is possible that
     * this pointer may not be the same as the one we sent to
     * usbdi_intfc_close_req().
     */
    printer->usbdi_intfc_open_close_cb = cb;

    /*
     * Did the close interface succeed?
     */
    if (status != UDI_OK) {
        /*
         * Handle failure cases
         */
        return;
    }

    /*
     * The interface is now closed.
     */
    printer->flags &= ~PRINT_FLAGS_GIO_UNBINDING;
    printer->flags |= PRINT_FLAGS_GIO_UNBOUND;

    udi_gio_unbind_ack(printer->gio_bind_cb);
    printer->gio_bind_cb = (udi_gio_bind_cb_t *)NULL;
}


/*
 * Function:  print_usbdi_intfc_abort_ack()
 *
 * This routine is called buy the USBD when an interface is successfully aborted.
 * We don't abort any interfaces in this driver, though, so it never get's called.
 */
static void
print_usbdi_intfc_abort_ack(usbdi_misc_cb_t *cb)
{
}


/*
 * Function:  print_usbdi_desc_ack()
 *
 * Called in response to a usbdi_desc_req() call.
 */
static void
print_usbdi_desc_ack(usbdi_desc_cb_t *cb,
                    udi_status_t status)
{
    struct printer_context *printer = UDI_GCB(cb)->context;
    struct printer_usbdi_desc_cb_scratch *scratch = UDI_GCB(cb)->scratch;

    /*
     * If we are unable to read the descriptor, fail the binding of the
     * driver.
     */
    if (status != UDI_OK) {

        /*
         * We're here because we were unable to read the device descriptor or
         * we were unable to find an uni-directional interface.
         *
         * Fail the binding since it does not provide the needed support
         * for the device.
         *
         * Note - since this is just a sample driver it doesn't do
         * complete error handling.  In some cases it would make sense
         * to retry the request if the status is
         * UDI_STAT_NOT_RESPONDING.
         */
        udi_cb_free(UDI_GCB(cb));
        print_free_printer(printer);
        udi_complete_channel_event(printer->usbdi_channel_cb,
                                   UDI_STAT_CANNOT_BIND);
        printer->usbdi_channel_cb = (udi_channel_event_cb_t *)UDI_NULL;
        return;
    }

    /*
     * Save away the returned descriptor.  We now own the memory for
     * this descriptor and we are responsible for releasing it when
     * we no longer need it.
     */
    *scratch->desc_buf = cb->desc_buf;

    /*
     * print_usbdi_desc_ack() is called on completion of any
     * usbdi_desc_req().  We may have just completed the read
     * of the device descriptor or we may have just completed the
     * read of an interface descriptor.
     * print_usbdi_read_interface_descriptor() will check to see
     * if the interface descriptor has already been retrieved.
     */
    print_usbdi_read_interface_descriptor(cb);
}


/*
 * Function:  print_usbdi_async_event_ind()
 *
 * This routine is called by the USBD when asynchronous events occur on the interface
 * channel.
 */
static void
print_usbdi_async_event_ind(usbdi_misc_cb_t *cb,
                           udi_ubit16_t async_event)
{
    switch (async_event) {

    case(USBDI_ASYNC_SUSPEND):
        break;

    case(USBDI_ASYNC_WAKEUP):
        break;

    case(USBDI_ASYNC_BANDWIDTH_NEEDED):
        break;

    default:
        break;
    }
}


/*
 * Function:  print_usbdi_ldd_pipe_channel_event_ind()
 *
 * This routine is the event indicator for USB pipes.  Right now, we shouldn't
 * need to deal with any of these, but we'll go ahead and recognize them
 * for the sake of being explicit.
 */
static void
print_usbdi_ldd_pipe_channel_event_ind(udi_channel_event_cb_t *cb)
{
    switch (cb->event) {
    case(UDI_CHANNEL_BOUND):
    case(UDI_CHANNEL_CLOSED):
    case(UDI_CHANNEL_OP_ABORTED):
    case(UDI_CONSTRAINTS_CHANGED):
    default:
        udi_channel_event_complete(cb, UDI_OK);
        break;
   }
}
```

```
/*
 * Function: print_usbdi_intr_bulk_xfer_ack()
 *
 * Called in response to a usbdi_intr_bulk_xfer() call.
 */
static void
print_usbdi_intr_bulk_xfer_ack(usbdi_intr_bulk_xfer_cb_t *bulk_cb)
{
    struct printer_pipe_context *pipe = UDI_GCB(bulk_cb)->context;
    struct printer_context *printer = pipe->printer;
    udi_gio_xfer_cb_t *gio_cb = UDI_GCB(bulk_cb)->context;

    /*
     * Update our usbdi_bulk_xfer_cb pointer since it may have
     * changed while it was owned by the USBD.
     */
    printer->bulk_xfer_cb = bulk_cb;

    /*
     * Update fields in our printer context struct.
     */
    printer->gio_xfer_cb = (udi_gio_xfer_cb_t *)NULL;
    printer->flags &= ~PRINT_FLAGS_GIO_XFER;

    udi_gio_xfer_ack(printer->gio_xfer_cb);

    /*
     * If the GIO client requested an unbind while this
     * USB request was in progress, we should now close the
     * interface
     */
    if (!(printer->flags & PRINT_FLAGS_GIO_BOUND)) {
        print_usbdi_intfc_close(UDI_GCB(printer->gio_bind_cb));
        udi_gio_unbind_ack(printer->gio_bind_cb);
    }

    return;
}


/*
 * Function: print_usbdi_intr_bulk_xfer_nak()
 *
 * Called on error in response to a usbdi_intr_bulk_xfer() call.
 */
static void
print_usbdi_intr_bulk_xfer_nak(usbdi_intr_bulk_xfer_cb_t *bulk_cb,
                               udi_status_t status)
{
    struct printer_pipe_context *pipe = UDI_GCB(bulk_cb)->context;
    struct printer_context *printer = pipe->printer;
    udi_gio_xfer_cb_t *gio_cb = printer->gio_xfer_cb;

    /*
     * Update our usbdi_bulk_xfer_cb pointer since it may have
     * changed while it was owned by the USBD.
     */
    printer->bulk_xfer_cb = bulk_cb;

    /*
     * Update fields in our printer context struct.
     */
    printer->gio_xfer_cb = (udi_gio_xfer_cb_t *)NULL;
    printer->flags &= ~PRINT_FLAGS_GIO_XFER;

    /*
     * Handle the error condition.
     */
    switch (status) {

    case USBDI_STAT_STALL:
        /*
         * We really shouldn't be getting a stall on the bulk out pipe.
         * Set the state of the endpoint to active.
         * Fall through to UDI_STAT_INVALID_STATE.
         */
        print_usbdi_edpt_state_set_req(pipe);
        break;

    case UDI_STAT_NOT_RESPONDING:
        /*
         * We may want to retry the request once or twice.
         *
         * For this sample driver we will pass the status on up.
         * Fall through to UDI_STAT_INVALID_STATE.
         */

    case UDI_STAT_ABORTED:
    case UDI_STAT_DATA_OVERRUN:
    case UDI_STAT_DATA_UNDERRUN:
    case UDI_STAT_TIMEOUT:
    case UDI_STAT_DATA_ERROR:
    case UDI_STAT_MISTAKEN_IDENTITY:
        /*
         * Need to add support for these error conditions.  For now,
         * fall through to UDI_STAT_INVALID_STATE.
         */

    case UDI_STAT_INVALID_STATE:
        default:
        /*
         * The pipe is in the USBDI_PIPE_IDLE state.  This will be the case
         * if the device is being deconfigured.  Don't try to change the
         * state of the pipe and don't retry the request.  Pass the error
         * on up.
         */
        gio_cb->data_buf->buf_size = UDI_NULL;
        udi_gio_xfer_nak(printer->gio_xfer_cb, status);
    }

    /*
     * If the GIO client requested an unbind while this
     * USB request was in progress, we should now close the
     * interface.  Otherwise, try to reset the pipe state.
     */
    if (!(printer->flags & PRINT_FLAGS_GIO_BOUND)) {

        print_usbdi_intfc_close(UDI_GCB(printer->gio_bind_cb));
        udi_gio_unbind_ack(printer->gio_bind_cb);
    }
    else {

        /*
         * In all other error cases the state of the pipe will be
         * USBDI_PIPE_STALLED.  Move the pipe to the active state.
         */
        print_usbdi_pipe_active_req(pipe);
    }

}

static void
print_usbdi_control_xfer_ack(usbdi_control_xfer_cb_t *cb,
                            udi_status_t status)
{
}


/*
 * Example of Opening an Interface
 *
 * The following functions open an interface.  This involves the allocation
 * of USB bandwidth by the USBD and creating channels for the pipes
 * associated with the interface.
 */

/*
 * print_usbdi_intfc_open()
 *
 * This function will be called print_gio_bin_req() when the
 * interface bound to this printer driver instance is requested to be
 * opened.  The final result will be that all pipes contained by the
 * interface will be opened and in the active state (ready to accept
 * transfer requests).
 *
 * Assumes gcb->context is set to our printer_context struct.
 */
static void
print_usbdi_intfc_open(udi_cb_t *gcb)
{
    struct printer_context *printer = gcb->context;

    /*
     * Do we need to allocate a usbdi_misc_cb_t?  If this is the
     * first open then we will need to allocate one, after that we
     * will reuse it.
     */
    if (printer->usbdi_intfc_open_close_cb ==
        (usbdi_misc_cb_t *)NULL) {
        udi_cb_alloc(print_usbdi_intfc_open_cb_alloc_call,
                     gcb,
                     PRINT_USBDI_OPEN_CLOSE_CB_IDX,
                     gcb->channel);
        return;

        /*
         * Execution continues in print_usbdi_intfc_open_cb_alloc_call().
         */
    }

    /*
     * We have the needed CB.  Request the open of the interface
     */
    usbdi_intfc_open_req(printer->usbdi_intfc_open_close_cb,
                         printer->usb_interface_descriptor->bAlternateSetting,
                         0);

    /*
     * Execution continues in print_usbdi_intfc_open_ack().
     */
}

/*
 * Function:  print_usbdi_intfc_open_cb_alloc_call()
 *
 * Callback function for alloc of usbdi_misc_cb_t
 * made in print_usbdi_intfc_open().
 */
static void
print_usbdi_intfc_open_cb_alloc_call(udi_cb_t *gcb,
                                     udi_cb_t *new_cb)
{
    struct printer_context *printer = gcb->context;

    printer->usbdi_intfc_open_close_cb = UDI_MCB(new_cb, usbdi_misc_cb_t);
    print_usbdi_intfc_open(gcb);
}

/*
 * Function: print_usbdi_intfc_open_ack()
 *
 * This function was given as the interface open op in
 * usbdi_ldd_intfc_ops_t.  It is called in response to a
 * usbdi_intfc_open_req() call.
 */
static void
print_usbdi_intfc_open_ack(usbdi_misc_cb_t *cb,
                           udi_index_t n_edpt, /* Interface endpoint count */
                           udi_status_t status)
{
    struct printer_context *printer = UDI_GCB(cb)->context;
    struct printer_usbdi_intfc_cb_scratch *scratch = UDI_GCB(cb)->scratch;

    /*
     * Update our usbdi_intfc_open_close_cb pointer.  It is possible that
     * this pointer may not be the same as the one we sent to
     * usbdi_intfc_open_req().
     */
    printer->usbdi_intfc_open_close_cb = cb;

    /*
     * Did the open interface succeed?
     */
    if (status != UDI_OK) {

        /*
         * Handle fail case
         */
        return;
    }

    /*
     * At this point the USBD has opened the interface and allocated
     * the needed bandwidth for all of the pipes and created the
     * USBD end of the channels for the endpoints.
     *
     * The LDD shall now complete the binding for channels to the endpoints.
     * print_usbdi_pipe_channel_spawn() will be called for each pipe.
     */

    /*
     * If this is the first open since the driver was bound then
     * intfc_edpt_count will be zero.  If this is not the first open
     * verify that intfc_edpt_count is the same as n_edpt.
     */
    if (printer->intfc_edpt_count == 0) {
        printer->intfc_edpt_count = n_edpt;

        /*
         * Allocate the array to hold our endpoint context array.
         */
        udi_mem_alloc(print_usbdi_alloc_edpt_context_call, UDI_GCB(cb),
                     sizeof(struct printer_pipe_context) * n_edpt, 0);
        return;

        /*
         * Execution continues in print_usbdi_alloc_edpt_context_call().
         */

    }
    else if (printer->intfc_edpt_count != n_edpt) {

        /*
         * The endpoint count doesn't match.
         *
         * Remember that this is just a sample driver so we
         * don't have to handle every little error :-).  Lazy, I know.
         */
        return;
    }

    /*
     * Initialize the pipe_spawn_count
     */
    scratch->pipe_spawn_count = n_edpt;

    /*
     * Start opening the channels for the pipes.
     */
    print_usbdi_pipe_channel_spawn(UDI_GCB(cb), printer);
}

/*
 * Function:  print_usbdi_alloc_edpt_context_call()
 *
 * Callback function given to udi_mem_alloc() in
 * print_usbdi_intfc_open_ack().
 */
static void
print_usbdi_alloc_edpt_context_call(udi_cb_t *gcb, void *new_mem)
{
    struct printer_context *printer = gcb->context;
    struct printer_usbdi_intfc_cb_scratch *scratch = gcb->scratch;
    int i;

    /*
     * Set up the array of endpoints
     */
    printer->pipe_context_array = new_mem;
    for (i=0; i < printer->intfc_edpt_count; i++) {
       printer->pipe_context_array[i].printer = printer;
    }

    /*
     * Initialize the pipe_spawn_count.  This is used by
     * print_usbdi_pipe_channel_spawn() to determine what pipe
     * the channel may be spawned for.
     */
    scratch->pipe_spawn_count = printer->intfc_edpt_count;

    /*
     * Start opening the channels for the pipes.
     */
    print_usbdi_pipe_channel_spawn(gcb, printer);
}

static void
print_usbdi_pipe_channel_spawn(udi_cb_t *gcb,
                              struct printer_context *printer)
{
    struct printer_usbdi_intfc_cb_scratch *scratch = gcb->scratch;
    udi_index_t endpoint = scratch->pipe_spawn_count;

    udi_channel_spawn(print_usbdi_pipe_channel_spawn_call, gcb,
                      printer->usbdi_intfc_channel, endpoint,
                      PRINT_USBDI_PIPE_OPS_IDX,
                      &printer->pipe_context_array[endpoint - 1]);

    /*
     * Execution continues in print_usbdi_pipe_channel_spawn_call()
     */
}

/*
 * Function:  print_usbdi_pipe_channel_spawn_call()
 *
 * Callback function given to udi_channel_spawn in
 * print_usbdi_pipe_channel_spawn().
 */
static void
print_usbdi_pipe_channel_spawn_call(udi_cb_t *gcb,
                                   udi_channel_t new_channel)
{
    struct printer_context *printer = gcb->context;
    struct printer_usbdi_intfc_cb_scratch *scratch = gcb->scratch;
    udi_index_t endpoint = --scratch->pipe_spawn_count;

    /*
     * Store away the newly allocated channel.
     */
    printer->pipe_context_array[endpoint].channel = new_channel;

    /*
     * Are there more endpoints that need channels?
     */
    if (endpoint > 0) {

        print_usbdi_pipe_channel_spawn(gcb, printer);

    } else {
        /*
         * All pipe channels have been spawned.
         * Call print_usbdi_intfc_open_complete().
         */
        print_usbdi_intfc_open_complete(printer);
    }
}

/*
 * Function: print_usbdi_intfc_open_complete()
 *
 * Called once all pipe channels for the interface have
 * been created.
 */
static void
print_usbdi_intfc_open_complete(struct printer_context *printer)
{
    /*
     * Have we already allocated our bulk_xfer_cb?
     */
    if (printer->bulk_xfer_cb == (usbdi_intr_bulk_xfer_cb_t *)NULL) {

        /*
         * Allocate our intr_bulk_xfer cb.  This CB is only allocated
         * on the first open of the driver and is then reused for each
         * operation associated with the current gio bind channel.
         */
        udi_cb_alloc(print_usbdi_intr_bulk_cb_alloc_call,
                     UDI_GCB(printer->gio_bind_cb),
                     PRINT_USBDI_BULK_XFER_CB_IDX,
                     UDI_GCB(printer->gio_bind_cb)->channel);
        return;

        /*
         * Execution continues with print_usbdi_intr_bulk_cb_alloc_call().
         */
    }

    /*
     * The open is complete.
     *
     * Notify the caller of print_gio_bind_req() that the interface
     * is now opened.
     */
    udi_gio_bind_ack(printer->gio_bind_cb, 0, 0, UDI_OK);
}

/*
 * Function: print_usbdi_intfc_close()
 *
 * This function is called when the interface has been
 * requested to be closed and all activity on all pipes
 * has completed.
 */
static void
print_usbdi_intfc_close(udi_cb_t *gcb)
```

```
{
    struct printer_context *printer = gcb->context;
    int i;

    /*
     * Close all channels that were spawned for the pipes of the interface.
     */
    for (i = 0; i < printer->intfc_edpt_count; i++) {
       udi_channel_close(printer->pipe_context_array[i].channel);
    }

    /*
     * Note - Don't free pipe_context_array.  It will be reused
     * if the interface is opened again.
     */

    /*
     * Note - usbdi_intfc_close() uses the same CB as usbdi_intfc_open()
     * therefore we don't have to allocate another CB.
     */

    /*
     * Request the close of the interface.
     */
    usbdi_intfc_close_req(printer->usbdi_intfc_open_close_cb);

    /*
     * Execution continues in print_usbdi_intfc_close_ack().
     */
}

/*
 * Example of an OpenUSBDI Bulk pipe transfer.
 */

/*
 * Function: print_usbdi_intr_bulk_cb_alloc_call()
 *
 * Callback function given to udi_cb_alloc to allocate a
 * usbdi_intr_bulk_cb.
 */
static void
print_usbdi_intr_bulk_cb_alloc_call(udi_cb_t *gcb,
                                  udi_cb_t *bulk_cb)
{
    struct printer_context *printer = gcb->context;

    /*
     * Save the usbdi_intr_bulk_xfer_cb_t away.
     */
    printer->bulk_xfer_cb = UDI_MCB(bulk_cb, usbdi_intr_bulk_xfer_cb_t);

    /*
     * Set the context pointer for the bulk_cb to the pipe_context
     * for the bulk pipe.
     */
    bulk_cb->context = &printer->pipe_context_array[printer->bulk_out_pipe];

    /*
     * Set up some of the fields in the CB that wont be changing.
     *
     * Set the timeout value to infinite and set the flags to zero.  These
     * will remain the same for all transfers that this driver will perform.
     */
    printer->bulk_xfer_cb->timeout = 0;
    printer->bulk_xfer_cb->xfer_flags = 0;

    /*
     * Allow the open of the interface to complete.
     */
    print_usbdi_intfc_open_complete(printer);
```

```
}

/*
 * Pipe state functions:
 *
 *    print_usbdi_pipe_active_req() - Set the state of the pipe to active.
 *
 *    print_usbdi_pipe_state_set_ack() - Ack called by the USBD once the
 *        state of the pipe has been set or if it couldn't be set.
 *
 *    print_usbdi_pipe_state_cb_alloc_call() - Called by the USBD once a
 *        usbdi_pipe_state_cb_t is available.
 */
static void
print_usbdi_pipe_active_req(struct printer_pipe_context *pipe)
{
    struct printer_context *printer = pipe->printer;
    usbdi_state_cb_t *cb = printer->usbdi_pipe_state_cb;

    /*
     * Have we already allocated a usbdi_pipe_state_cb?
     */
    if (cb == (usbdi_state_cb_t *)NULL) {
        udi_cb_alloc(print_usbdi_pipe_state_cb_alloc_call,
                     UDI_GCB(printer->bulk_xfer_cb),
                     PRINT_USBDI_STATE_CB_IDX,
                     pipe->channel);
        return;

        /*
         * Execution continues in print_usbdi_pipe_state_cb_alloc_call().
         */
    }

    cb->state = USBDI_STATE_ACTIVE;
    usbdi_pipe_state_set_req(cb);

    /*
     * Execution continues in print_usbdi_pipe_state_set_ack().
     */
}

/*
 * Function: print_usbdi_pipe_state_set_ack()
 *
 */
static void
print_usbdi_pipe_state_set_ack(usbdi_state_cb_t *cb,
                               udi_status_t status)
{
    struct printer_pipe_context *pipe = UDI_GCB(cb)->context;
    struct printer_context *printer = pipe->printer;

    /*
     * Update our usbdi_pipe_state_cb pointer since it may have changed
     * while the USBD had owned it.
     */
    printer->usbdi_pipe_state_cb = cb;

    /*
     * Check our status.
     */
    switch (status) {

    case UDI_OK:
        /*
         * The pipe state was successfully set.
         */

    case UDI_STAT_INVALID_STATE:
        /*
```

```
         * The pipe state was not set.  This is due to the state of the
         * interface being something other than USBDI_INTFC_ACTIVE.  This
         * may be the case if the device is in the process of being
         * deconfigured.
         */

    case UDI_STAT_MISTAKEN_IDENTITY:
        /*
         * Neither of these two cases may be happening.
         */

    default:
        /*
         * In all cases, do nothing.
         */
        break;
    }
}

static void
print_usbdi_pipe_state_cb_alloc_call(udi_cb_t *gcb,
                                   udi_cb_t *cb)
{
    struct printer_pipe_context *pipe = gcb->context;
    struct printer_context *printer = pipe->printer;
    printer->usbdi_pipe_state_cb = UDI_MCB(cb, usbdi_state_cb_t);
    print_usbdi_pipe_active_req(pipe);
}

/*
 * Endpoint state functions:
 *
 *    print_usbdi_edpt_state_set_req() - Set endpoint state to active.
 *
 *    print_usbdi_edpt_state_cb_alloc_call() - Called by the USBD once a
 *        usbdi_edpt_state_cb_t is available.
 *
 *    print_usbdi_edpt_state_set_ack() - Ack called by the USBD once the
 *        state of the endpoint has been set or if it couldn't be set.
 */
static void
print_usbdi_edpt_state_set_req(struct printer_pipe_context *pipe)
{
    struct printer_context *printer = pipe->printer;

    /*
     * Allocated a usbdi_edpt_state_cb.  We don't expect to be
     * setting the state of the endpoint very often, so don't
     * keep the CB around after the request completes.
     */
    udi_cb_alloc(print_usbdi_edpt_state_cb_alloc_call,
                UDI_GCB(printer->bulk_xfer_cb),
                PRINT_USBDI_STATE_CB_IDX,
                pipe->channel);
}

static void
print_usbdi_edpt_state_cb_alloc_call(udi_cb_t *gcb, udi_cb_t *cb)
{
    struct printer_pipe_context *pipe = gcb->context;
    struct printer_context *printer = pipe->printer;
    usbdi_state_cb_t *edpt_state_cb = UDI_MCB(cb, usbdi_state_cb_t);

    edpt_state_cb->gcb.context = pipe;
    edpt_state_cb->state = USBDI_STATE_ACTIVE;
    usbdi_edpt_state_set_req(edpt_state_cb);
}

static void
print_usbdi_edpt_state_set_ack(usbdi_state_cb_t *cb,
                              udi_status_t status)
```

```
{
    struct printer_pipe_context *pipe = UDI_GCB(cb)->context;
    struct printer_context *printer = pipe->printer;

    /*
     * Check our status.
     */
    switch (status) {

    case UDI_OK:
        /*
         * The endpoint state was successfully set.
         */

    case UDI_STAT_MISTAKEN_IDENTITY:
    default:
        /*
         * In all cases, do nothing.
         */
        break;
    }

    /*
     * Release the cb.
     */
    udi_cb_free(UDI_GCB(cb));
}

/*
 * Function:  print_free_printer()
 *
 * Do some final cleanup.  The bind and channel control blocks are not
 * touched by this routine.
 */
static void
print_free_printer(struct printer_context *printer)
{
    udi_mem_free(printer->pipe_context_array);
    printer->pipe_context_array = (struct printer_pipe_context *)NULL;
    udi_mem_free(printer->usb_device_descriptor);
    printer->usb_device_descriptor = (struct usb_device_descriptor *)NULL;
    udi_mem_free(printer->usb_interface_descriptor);
    printer->usb_interface_descriptor =(struct usb_interface_descriptor *)NULL;
    udi_cb_free(UDI_GCB(printer->usbdi_intfc_open_close_cb));
    printer->usbdi_intfc_open_close_cb = (usbdi_misc_cb_t *)NULL;
    udi_cb_free(UDI_GCB(printer->bulk_xfer_cb));
    printer->bulk_xfer_cb = (usbdi_intr_bulk_xfer_cb_t *)NULL;
    udi_cb_free(UDI_GCB(printer->usbdi_pipe_state_cb));
    printer->usbdi_pipe_state_cb = (usbdi_state_cb_t *)NULL;

    /*
     * Add unbind from USBD.
     */
}

static void
print_usbdi_pipe_state_get_ack(usbdi_state_cb_t *cb)
{
}

static void
print_usbdi_edpt_state_get_ack(usbdi_state_cb_t *cb)
{
}

/*
 * UDI Management Ops
 */
static void
print_udi_devmgmt_req(udi_mgmt_cb_t *cb, udi_ubit8_t mgmt_op,
                     udi_ubit8_t parent_ID)
```

```
{
    switch(mgmt_op)
    {
    case(UDI_DMGMT_PREPARE_TO_SUSPEND):
    case(UDI_DMGMT_SUSPEND):
    case(UDI_DMGMT_SHUTDOWN):
    case(UDI_DMGMT_PARENT_SUSPENDED):
    case(UDI_DMGMT_RESUME):
    case(UDI_DMGMT_UNBIND):
    default:
       udi_devmgmt_ack(cb, UDI_NULL, UDI_STAT_NOT_SUPPORTED);
       break;
    }
}

/*
 * This call should cause us to prepare for shutdown or unloading
 */
static void
print_udi_final_cleanup_req(udi_mgmt_cb_t* cb)
{
    /*
     * Do cleanup, then respond with:
     */
    udi_final_cleanup_ack(cb);
}
```

9 Appendix C: Sample Driver (usbdi_printer.h)

```
/*
 * USB Printer Class defines
 */
#define INTERFACE_CLASS_PRINTER                      0x07
#define INTERFACE_SUBCLASS_PRINTER                   0x01
#define INTERFACE_PROTOCOL_PRINTER_UNIDIRECTIONAL    0x01
#define INTERFACE_PROTOCOL_PRINTER_BIDIRECTIONAL     0x02

/*
 * This file contains all structures, defines, and declarations for
 * the OpenUSBDI uni-directional printer class driver.
 */
struct printer_context; /* Forward declaration */

/*
 * Printer specific pipe context structure.  One per pipe.  This
 * LDD instance will only need one of these structures since it
 * will only be working with a single bulk out endpoint.
 */
struct printer_pipe_context {
    struct printer_context *printer; /* Pointer to the associated printer */
    udi_channel_t channel;           /* UDI channel for this pipe */
};

/*
 * Printer context structure
 */
struct printer_context {

    udi_init_context_t init_context;

    void *enumeration_context;

    /* These are some channels that this module will use.
     *
     * They need to be placed globally because the channel_event_ind routines get NO
     * context or scratch space in their input CBs.  This is important because when we
     * get called in those routines, we are expected to close channels with
     * udi_channel_close() (which requires a channel).
     */

    udi_channel_t usbdi_pipe_channel; /* First channel given to us when the MA
                                       * initializes the pipe channel.
                                       */
    udi_channel_t usbdi_intfc_channel;/* Channel used to communicate with the
                                       * USBD when performing intfc related
                                       * ops
                                       */
    udi_channel_event_cb_t *usbdi_channel_cb; /* CB given to us by the MA
                                               * when the interface channel
                                               * is bound
                                               */

    udi_gio_bind_cb_t *gio_bind_cb; /* CB given to print_gio_bind_req() and
                                     * print_gio_unbind_req()
                                     */
    usbdi_misc_cb_t *usbdi_bind_cb; /* CB used to bind and unbind with USBD */

    udi_index_t intfc_edpt_count;

    struct printer_pipe_context *pipe_context_array;
    udi_index_t bulk_out_pipe;      /* index into pipe_context_array for our
                                     * bulk out pipe.
                                     */

    /*
     * GIO bind fields
     */
    udi_channel_t bind_target_channel;

    /*
     * USB descriptor pointers.
     */
    struct usb_device_descriptor *usb_device_descriptor;
    struct usb_interface_descriptor *usb_interface_descriptor;

    /*
     * Open interface fields
     */
    usbdi_misc_cb_t *usbdi_intfc_open_close_cb;

    /*
     * Transfer fields.  This driver was written to only allow
     * one outstanding usbdi_intr_bulk_xfer_cb_t at a time.
     */
    udi_gio_xfer_cb_t *gio_xfer_cb;
    usbdi_intr_bulk_xfer_cb_t *bulk_xfer_cb;

    /*
     * Pipe state
     */
    usbdi_state_cb_t *usbdi_pipe_state_cb;

    udi_ubit32_t flags;
#define PRINT_FLAGS_DESC_BUSY         (1<<0)
#define PRINT_FLAGS_GIO_BOUND         (1<<1)
#define PRINT_FLAGS_GIO_UNBOUND       (1<<2)
#define PRINT_FLAGS_GIO_UNBINDING     (1<<3)
#define PRINT_FLAGS_GIO_XFER          (1<<4)
};

/*
 * USBDI LDD scratch structures.
 *
 * Scratch struct for usbdi_desc_cb_t
 */
struct printer_usbdi_desc_cb_scratch {
    udi_buf_t *desc_buf;          /* Where to put the descriptor */
};

/*
 * Scratch struct for usbdi_intfc_open_close
 */
struct printer_usbdi_intfc_cb_scratch {
    udi_index_t pipe_spawn_count; /* What pipe do we spawn ? */
};

/*
 * This driver assumes that is will be bound to PRINTER_INTFC_COUNT
 * number of interfaces.
 */
#define PRINTER_INTFC_COUNT 1

/*
 * udi_cb_alloc() callback functions
 */
static udi_cb_alloc_call_t print_usbdi_bind_cb_alloc_call;
static udi_cb_alloc_call_t print_usbdi_desc_cb_alloc_call;
static udi_cb_alloc_call_t print_usbdi_pipe_state_cb_alloc_call;
static udi_cb_alloc_call_t print_usbdi_intr_bulk_cb_alloc_call;
static udi_cb_alloc_call_t print_usbdi_edpt_state_cb_alloc_call;
static udi_cb_alloc_call_t print_usbdi_intfc_open_cb_alloc_call;

/*
 * udi_mem_alloc() callback functions
 */
static udi_mem_alloc_call_t print_usbdi_alloc_edpt_context_call;
```

```
/*
 * udi_channel_spawn() callback functions
 */
static udi_channel_spawn_call_t print_usbdi_pipe_channel_spawn_call;

/*
 * Local functions
 */
static void print_usbdi_read_descriptors(struct printer_context *printer);
static void print_usbdi_pipe_channel_spawn(udi_cb_t *gcb,
                                        struct printer_context *printer);
static void print_usbdi_intfc_close(udi_cb_t *gcb);
static void print_free_printer(struct printer_context *printer) ;
static void print_usbdi_edpt_state_set_req(struct printer_pipe_context *pipe);
static void print_usbdi_pipe_active_req(struct printer_pipe_context *pipe);
static void print_usbdi_bulk_cb_alloc(struct printer_context *printer);
static void print_usbdi_intfc_open(udi_cb_t *gcb);
static void print_usbdi_intfc_open_complete(struct printer_context *printer);

/*
 * udi_mgmt_ops_t
 */
udi_usage_ind_op_t udi_static_usage;                /* Proxy */
udi_enumerate_req_op_t udi_enumerate_no_children;   /* Proxy */
static void print_udi_devmgmt_req(udi_mgmt_cb_t *cb, udi_ubit8_t mgmt_op,
                                udi_ubit8_t parent_ID);
static void print_udi_final_cleanup_req(udi_mgmt_cb_t *cb);
extern void udi_final_cleanup_ack(udi_mgmt_cb_t *cb);

/*
 * usbdi_ldd_pipe_ops_t
 */
static udi_channel_event_ind_op_t     print_usbdi_ldd_pipe_channel_event_ind;
static usbdi_intr_bulk_xfer_ack_op_t  print_usbdi_intr_bulk_xfer_ack;
static usbdi_intr_bulk_xfer_nak_op_t  print_usbdi_intr_bulk_xfer_nak;
static usbdi_control_xfer_ack_op_t    print_usbdi_control_xfer_ack;
static usbdi_pipe_state_set_ack_op_t  print_usbdi_pipe_state_set_ack;
static usbdi_pipe_state_get_ack_op_t  print_usbdi_pipe_state_get_ack;
static usbdi_edpt_state_set_ack_op_t  print_usbdi_edpt_state_set_ack;
static usbdi_edpt_state_get_ack_op_t  print_usbdi_edpt_state_get_ack;

/*
 * usbdi_ldd_intfc_ops_t
 */
static udi_channel_event_ind_op_t     print_usbdi_ldd_intfc_channel_event_ind;
static usbdi_bind_ack_op_t            print_usbdi_bind_ack;
static usbdi_unbind_ack_op_t          print_usbdi_unbind_ack;
static usbdi_intfc_open_ack_op_t      print_usbdi_intfc_open_ack;
static usbdi_intfc_close_ack_op_t     print_usbdi_intfc_close_ack;
static usbdi_intfc_abort_ack_op_t     print_usbdi_intfc_abort_ack;
static usbdi_desc_ack_op_t            print_usbdi_desc_ack;
static usbdi_async_event_ind_op_t     print_usbdi_async_event_ind;

/*
 * udi_gio_provider_ops_t
 */
static udi_channel_event_ind_op_t     print_gio_provider_channel_event_ind;
static udi_gio_bind_req_op_t          print_gio_bind_req;
static udi_gio_unbind_req_op_t        print_gio_unbind_req;
static udi_gio_xfer_req_op_t          print_gio_xfer_req;


/*
 * Index values for all CB's that this driver will allocate.
 *
 * A driver that allocates the same CB with different scratch
 * sizes would have to provide a different index values for
 * each scratch size.
 */
#define PRINT_USBDI_INTFC_OPS_IDX           1
#define PRINT_USBDI_PIPE_OPS_IDX            2
```

```
#define PRINT_USBDI_MISC_CB_IDX              1
#define PRINT_USBDI_BULK_XFER_CB_IDX         2
#define PRINT_USBDI_CONTROL_XFER_CB_IDX      3
#define PRINT_USBDI_STATE_CB_IDX             4
#define PRINT_USBDI_DESC_CB_IDX              5
#define PRINT_USBDI_OPEN_CLOSE_CB_IDX        6

/*
 * Scratch sizes for each of the USBDI CBs.
 */
#define PRINT_USBDI_MISC_CB_SCRATCH_SIZE            0
#define PRINT_USBDI_BULK_XFER_CB_SCRATCH_SIZE       0
#define PRINT_USBDI_CONTROL_XFER_CB_SCRATCH_SIZE    0
#define PRINT_USBDI_STATE_CB_SCRATCH_SIZE           0
#define PRINT_USBDI_DESC_CB_SCRATCH_SIZE            \
     sizeof(struct printer_usbdi_desc_cb_scratch)
#define PRINT_USBDI_OPEN_CLOSE_CB_SCRATCH_SIZE      \
     sizeof(struct printer_usbdi_intfc_cb_scratch)

/*
 * Printer driver specific GIO indexes
 */
#define PRINT_GIO_PROVIDER_OPS_IDX           1
#define PRINT_GIO_BIND_CB_IDX                1
#define PRINT_GIO_UNBIND_CB_IDX              2
#define PRINT_GIO_XFER_CB_IDX                3
#define PRINT_GIO_EVENT_CB_IDX               4

/*
 * Printer driver specific GIO scratch sizes
 */
#define PRINT_GIO_BIND_CB_SCRATCH_SIZE       sizeof(void *)
#define PRINT_GIO_UNBIND_CB_SCRATCH_SIZE     sizeof(void *)
#define PRINT_GIO_XFER_CB_SCRATCH_SIZE       sizeof(void *)
#define PRINT_GIO_XFER_CB_PARAMS_SIZE        sizeof(void *)
#define PRINT_GIO_EVENT_CB_SCRATCH_SIZE      sizeof(void *)

/*
 * Primary region size
 */
#define PRINT_UDI_PRIMARY_REGION_SIZE        \
    (sizeof(udi_init_context_t) + sizeof( struct printer_context))

/*
 * Identify the metalanguages
 */
#define PRINT_USBDI_META_IDX       1
#define PRINT_GIO_META_IDX         2
```

10 Appendix D: FCS Computation Table

32-bit FCS Computation Method

The following code provides a table lookup computation for calculating the 32-bit Frame Check Sequence.

```
/*
 * The FCS-32 generator polynomial: x**0 + x**1 + x**2 + x**4 + x**5
 *   + x**7 + x**8 + x**10 + x**11 + x**12 + x**16
 *   + x**22 + x**23 + x**26 + x**32.
 */


static u32 fcstab_32[256] =
{
0x00000000, 0x77073096, 0xee0e612c, 0x990951ba,
0x076dc419, 0x706af48f, 0xe963a535, 0x9e6495a3,
0x0edb8832, 0x79dcb8a4, 0xe0d5e91e, 0x97d2d988,
0x09b64c2b, 0x7eb17cbd, 0xe7b82d07, 0x90bf1d91,
0x1db71064, 0x6ab020f2, 0xf3b97148, 0x84be41de,
0x1adad47d, 0x6ddde4eb, 0xf4d4b551, 0x83d385c7,
0x136c9856, 0x646ba8c0, 0xfd62f97a, 0x8a65c9ec,
0x14015c4f, 0x63066cd9, 0xfa0f3d63, 0x8d080df5,
0x3b6e20c8, 0x4c69105e, 0xd56041e4, 0xa2677172,
0x3c03e4d1, 0x4b04d447, 0xd20d85fd, 0xa50ab56b,
0x35b5a8fa, 0x42b2986c, 0xdbbbc9d6, 0xacbcf940,
0x32d86ce3, 0x45df5c75, 0xdcd60dcf, 0xabd13d59,
0x26d930ac, 0x51de003a, 0xc8d75180, 0xbfd06116,
0x21b4f4b5, 0x56b3c423, 0xcfba9599, 0xb8bda50f,
0x2802b89e, 0x5f058808, 0xc60cd9b2, 0xb10be924,
0x2f6f7c87, 0x58684c11, 0xc1611dab, 0xb6662d3d,
0x76dc4190, 0x01db7106, 0x98d220bc, 0xefd5102a,
0x71b18589, 0x06b6b51f, 0x9fbfe4a5, 0xe8b8d433,
0x7807c9a2, 0x0f00f934, 0x9609a88e, 0xe10e9818,
0x7f6a0dbb, 0x086d3d2d, 0x91646c97, 0xe6635c01,
0x6b6b51f4, 0x1c6c6162, 0x856530d8, 0xf262004e,
0x6c0695ed, 0x1b01a57b, 0x8208f4c1, 0xf50fc457,
0x65b0d9c6, 0x12b7e950, 0x8bbeb8ea, 0xfcb9887c,
0x62dd1ddf, 0x15da2d49, 0x8cd37cf3, 0xfbd44c65,
0x4db26158, 0x3ab551ce, 0xa3bc0074, 0xd4bb30e2,
0x4adfa541, 0x3dd895d7, 0xa4d1c46d, 0xd3d6f4fb,
0x4369e96a, 0x346ed9fc, 0xad678846, 0xda60b8d0,
0x44042d73, 0x33031de5, 0xaa0a4c5f, 0xdd0d7cc9,
0x5005713c, 0x270241aa, 0xbe0b1010, 0xc90c2086,
0x5768b525, 0x206f85b3, 0xb966d409, 0xce61e49f,
0x5edef90e, 0x29d9c998, 0xb0d09822, 0xc7d7a8b4,
0x59b33d17, 0x2eb40d81, 0xb7bd5c3b, 0xc0ba6cad,
0xedb88320, 0x9abfb3b6, 0x03b6e20c, 0x74b1d29a,
0xead54739, 0x9dd277af, 0x04db2615, 0x73dc1683,
0xe3630b12, 0x94643b84, 0x0d6d6a3e, 0x7a6a5aa8,
0xe40ecf0b, 0x9309ff9d, 0x0a00ae27, 0x7d079eb1,
0xf00f9344, 0x8708a3d2, 0x1e01f268, 0x6906c2fe,
0xf762575d, 0x806567cb, 0x196c3671, 0x6e6b06e7,
0xfed41b76, 0x89d32be0, 0x10da7a5a, 0x67dd4acc,
0xf9b9df6f, 0x8ebeeff9, 0x17b7be43, 0x60b08ed5,
0xd6d6a3e8, 0xa1d1937e, 0x38d8c2c4, 0x4fdff252,
0xd1bb67f1, 0xa6bc5767, 0x3fb506dd, 0x48b2364b,
0xd80d2bda, 0xaf0a1b4c, 0x36034af6, 0x41047a60,
0xdf60efc3, 0xa867df55, 0x316e8eef, 0x4669be79,
0xcb61b38c, 0xbc66831a, 0x256fd2a0, 0x5268e236,
0xcc0c7795, 0xbb0b4703, 0x220216b9, 0x5505262f,
0xc5ba3bbe, 0xb2bd0b28, 0x2bb45a92, 0x5cb36a04,
0xc2d7ffa7, 0xb5d0cf31, 0x2cd99e8b, 0x5bdeae1d,
0x9b64c2b0, 0xec63f226, 0x756aa39c, 0x026d930a,
0x9c0906a9, 0xeb0e363f, 0x72076785, 0x05005713,
0x95bf4a82, 0xe2b87a14, 0x7bb12bae, 0x0cb61b38,
0x92d28e9b, 0xe5d5be0d, 0x7cdcefb7, 0x0bdbdf21,
0x86d3d2d4, 0xf1d4e242, 0x68ddb3f8, 0x1fda836e,
0x81be16cd, 0xf6b9265b, 0x6fb077e1, 0x18b74777,
0x88085ae6, 0xff0f6a70, 0x66063bca, 0x11010b5c,
0x8f659eff, 0xf862ae69, 0x616bffd3, 0x166ccf45,
0xa00ae278, 0xd70dd2ee, 0x4e048354, 0x3903b3c2,
0xa7672661, 0xd06016f7, 0x4969474d, 0x3e6e77db,
0xaed16a4a, 0xd9d65adc, 0x40df0b66, 0x37d83bf0,
0xa9bcae53, 0xdebb9ec5, 0x47b2cf7f, 0x30b5ffe9,
0xbdbdf21c, 0xcabac28a, 0x53b39330, 0x24b4a3a6,
0xbad03605, 0xcdd70693, 0x54de5729, 0x23d967bf,
0xb3667a2e, 0xc4614ab8, 0x5d681b02, 0x2a6f2b94,
0xb40bbe37, 0xc30c8ea1, 0x5a05df1b, 0x2d02ef8d
};

#define USBDI_INIT_FCS_32  0xffffffff   /* Initial FCS value */
/*
* Calculate a new FCS given the current FCS and the new data.
*/

udi_ubi32_t usbdi_fcs_32(register udi_ubit8_t *cp,
                         register udi_ubit32_t len)
{
    register ubit32_t fcs = USBDI_INIT_FCS_32;
    ASSERT(sizeof (u32) == 4);
    ASSERT(((u32) -1) > 0);
    while (len--)
    {
       fcs = (((fcs) >> 8) ^ fcstab_32[((fcs) ^ (*cp++)) & 0xff]);
    }

    return (fcs);

}
```