Open Universal Serial Bus Driver Interface (OpenUSBDI) Specification

Revision 0.9

June 21, 1999

Open USB Driver Interface (OpenUSBDI) Specification
© Compaq Computer Corporation, All rights reserved.

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Not for Publication or General Distribution.

Revision History

Rev	Date	Filename	Comments
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.

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Not For Publication or General Distribution.

INTELLECTUAL PROPERTY DISCLAIMER

THIS SPECIFICATION IS PROVIDED “AS IS” WITH NO WARRANTIES WHATSOEVER INCLUDING ANY WARRANTY OF MERCHANTABILITY, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION, OR SAMPLE.

A LICENSE IS HEREBY GRANTED TO REPRODUCE AND DISTRIBUTE THIS SPECIFICATION FOR INTERNAL USE ONLY. NO OTHER LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY OTHER INTELLECTUAL PROPERTY RIGHTS IS GRANTED OR INTENDED HEREBY.

AUTHORS OF THIS SPECIFICATION DISCLAIM ALL LIABILITY, INCLUDING LIABILITY FOR INFRINGEMENT OF PROPRIETARY RIGHTS, RELATING TO IMPLEMENTATION OF INFORMATION IN THIS SPECIFICATION. AUTHORS OF THIS SPECIFICATION ALSO DO NOT WARRANT OR REPRESENT THAT SUCH IMPLEMENTATION(S) WILL NOT INFRINGE SUCH RIGHTS.

All product names are trademarks, registered trademarks, or servicemarks of their respective owners.

Contributors

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

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Not for Publication or General Distribution.

Table of Contents

1 Introduction 9
1.1 Purpose 9
1.2 Scope 9
1.3 Related Documents 9
1.4 Terms and Abbreviations 9
2 Management Overview 13
3 Functional Overview 16
4 Functional Characteristics 18
4.1 Required Include Files 18
4.2 Channels 18
4.3 Completion Status Values 18
4.4 Device Driver Flow 18
4.4.1 Logical-Device Driver Initialization 20
4.4.2 Interface Binding 21
4.4.3 Setting the Configuration 22
4.4.4 Reading Descriptors 23
4.4.5 Opening an Interface 24
4.4.6 Closing an Interface 25
4.4.7 Transferring Data 26
4.4.8 Completing Requests 28
4.4.9 Pipe State Control 29
4.4.10 Interface State Control 30
4.4.11 Endpoint State Control 31
4.4.12 Aborting Transfer Requests 31
5 OpenUSBDI Metalanguage 33
5.1 Driver Registration 34
5.1.1 usbdi_ldd_intfc_ops_t 34
5.1.2 usbdi_ldd_intfc_ops_init() 35
5.1.3 usbdi_ldd_pipe_ops_t 36
5.1.4 usbdi_ldd_pipe_ops_init() 37
5.2 Miscellaneous Control Block 38
5.2.1 usbdi_misc_cb_t 38
5.2.2 usbdi_misc_cb_init() 39
5.3 Driver Binding 40
5.3.1 usbdi_bind_req() 41
5.3.2 usbdi_bind_ack_op_t 42
5.4 Driver Unbinding 43
5.4.1 usbdi_unbind_req() 44
5.4.2 usbdi_unbind_ack_op_t 45
5.5 Opening an Interface 46

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

5.5.1 usbdi_intfc_open_req() 47
5.5.2 usbdi_intfc_open_ack_op_t 48
5.6 Closing an Interface....50
5.6.1 usbdi_intfc_close_req()....51
5.6.2 usbdi_intfc_close_ack_op_t 52
5.7 USB Interrupt and Bulk Transfer Requests....53
5.7.1 usbdi_intr_bulk_xfer_cb_t 53
5.7.2 usbdi_intr_bulk_xfer_cb_init() 55
5.7.3 usbdi_intr_bulk_xfer_req() 56
5.7.4 usbdi_intr_bulk_xfer_ack_op_t....58
5.7.5 usbdi_intr_bulk_xfer_nak_op_t 59
5.7.6 usbdi_intr_bulk_xfer_ack_unused()....61
5.7.7 usbdi_intr_bulk_xfer_nak_unused()....61
5.8 USB Control Transfer Requests....62
5.8.1 usbdi_control_xfer_cb_t....62
5.8.2 usbdi_control_xfer_cb_init()....64
5.8.3 usbdi_control_xfer_req()....65
5.8.4 usbdi_control_xfer_ack_op_t 67
5.8.5 usbdi_control_xfer_ack_unused() 69
5.9 USB Isochronous Transfer Requests 70
5.9.1 usbdi_isoc_frame_request_t 71
5.9.2 usbdi_isoc_frame_request_t 71
5.9.3 usbdi_isoc_xfer_cb_t 72
5.9.4 usbdi_isoc_xfer_cb_init() 74
5.9.5 usbdi_isoc_xfer_req() 75
5.9.6 usbdi_isoc_xfer_ack_op_t....77
5.9.7 usbdi_isoc_xfer_nak_op_t 78
5.9.8 usbdi_isoc_xfer_ack_unused()....80
5.9.9 usbdi_isoc_xfer_nak_unused() 80
5.10 USB Isochronous Frame Number Determination....81
5.10.1 usbdi_frame_number_req()....82
5.10.2 usbdi_frame_number_ack_op_t....83
5.10.3 usbdi_frame_number_ack_unused()....84
5.11 Resetting a Device 85
5.11.1 usbdi_reset_device_req() 86
5.11.2 usbdi_reset_device_ack_op_t....87
5.11.3 usbdi_reset_device_ack_unused()....88
5.12 Aborting a Transfer....89
5.12.1 usbdi_xfer_abort_cb_t 89
5.12.2 usbdi_xfer_abort_cb_init() 90
5.12.3 usbdi_xfer_abort_req()....91
5.12.4 usbdi_xfer_abort_ack_op_t....92

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

5.12.5 usbdi_xfer_abort_ack_unused() 93
5.13 Aborting a Pipe 94
5.13.1 usbdi_pipe_abort_req() 95
5.13.2 usbdi_pipe_abort_ack_op_t 96
5.13.3 usbdi_pipe_abort_ack_unused() 97
5.14 Aborting an Interface 98
5.14.1 usbdi_intfc_abort_req() 99
5.14.2 usbdi_intfc_abort_ack_op_t 100
5.14.3 usbdi_intfc_abort_ack_unused() 101
5.15 Getting and Setting States 102
5.15.1 usbdi_state_cb_t 102
5.15.2 usbdi_state_cb_init() 103
5.16 Getting and Setting Pipe States 104
5.16.1 usbdi_pipe_state_set_req() 105
5.16.2 usbdi_pipe_state_set_ack_op_t 106
5.16.3 usbdi_pipe_state_get_req() 107
5.16.4 usbdi_pipe_state_get_ack_op_t 108
5.16.5 usbdi_pipe_state_get_ack_unused() 109
5.17 Getting and Setting Endpoint States 110
5.17.1 usbdi_edpt_state_set_req() 111
5.17.2 usbdi_edpt_state_set_ack_op_t 112
5.17.3 usbdi_edpt_state_get_req() 113
5.17.4 usbdi_edpt_state_get_ack_op_t 114
5.17.5 usbdi_edpt_state_get_ack_unused() 115
5.18 Getting and Setting Interface States 116
5.18.1 usbdi_intfc_state_set_req() 117
5.18.2 usbdi_intfc_state_set_ack_op_t 118
5.18.3 usbdi_intfc_state_set_ack_unused() 119
5.18.4 usbdi_intfc_state_get_req() 120
5.18.5 usbdi_intfc_state_get_ack_op_t 121
5.18.6 usbdi_intfc_state_get_ack_unused() 122
5.19 Getting Device States 123
5.19.1 usbdi_device_state_get_req() 124
5.19.2 usbdi_device_state_get_ack_op_t 125
5.19.3 usbdi_device_state_get_ack_unused() 126
5.20 Retrieving Descriptors 127
5.20.1 usbdi_desc_cb_t 128
5.20.2 usbdi_desc_cb_init() 129
5.20.3 usbdi_desc_req() 130
5.20.4 usbdi_desc_ack_op_t 131
5.20.5 usbdi_desc_ack_unused() 132
5.21 Changing a Device's Configuration 133

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

5.21.1 usbdi_config_set_req() 134
5.21.2 usbdi_config_set_ack_op_t 135
5.21.3 usbdi_config_set_ack_unused() 136
5.22 Asynchronous Events 137
5.22.1 usbdi_async_event_ind_op_t 138
5.22.2 usbdi_async_event_ind_unused() 139
6 UDI Overview 141
6.1 Management Agent 141
6.2 udi_cb_alloc() 142
6.3 udi_cb_free() 143
6.4 udi_channel_spawn() 144
6.5 udi_channel_close() 146
6.6 udi_cb_t 147
7 Helpful Hints 148
8 Outstanding Issues 149
9 Appendix A: Include File (open_usbdi.h) 150
10 Appendix B: Sample Driver (usbdi_printer.c) 169
11 Appendix C: Sample Driver (usbdi_printer.h) 191

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

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*, Rev 0.90, March 30, 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.)

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

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.

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

Term	Description
Logical-Device	One or more USB interfaces that function together as a single entity.
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.

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

Term	Description
UHCI	Universal Host Controller Interface
USB device	See “device”.
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.

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

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. See Section 6, "UDI Overview", for more information on UDI.

For more information on "Project UDI" see the UDI web page, http://www.sco.com/udi.

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

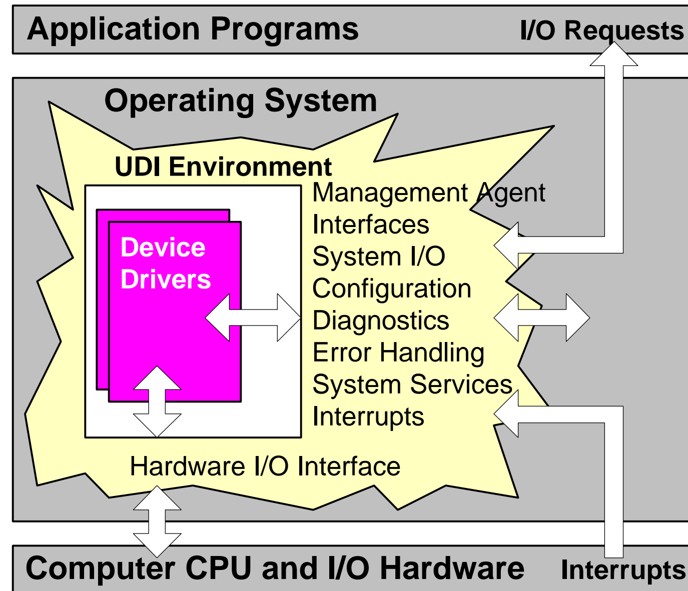

Figure 1: UDI Driver Environment

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

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.

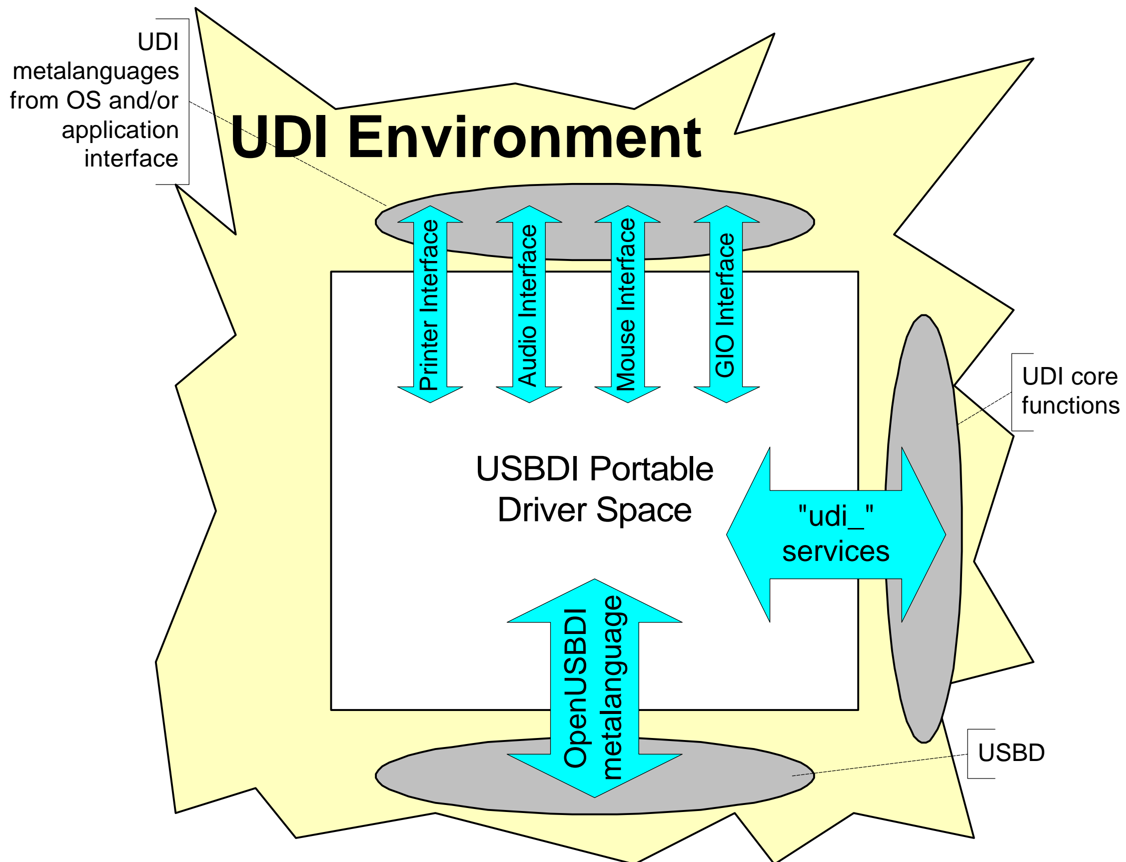

Figure 2: OpenUSBDI Portable Environment

[1] Defined in the UDI Core Specification

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

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 device driver 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 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.

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

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 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).

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

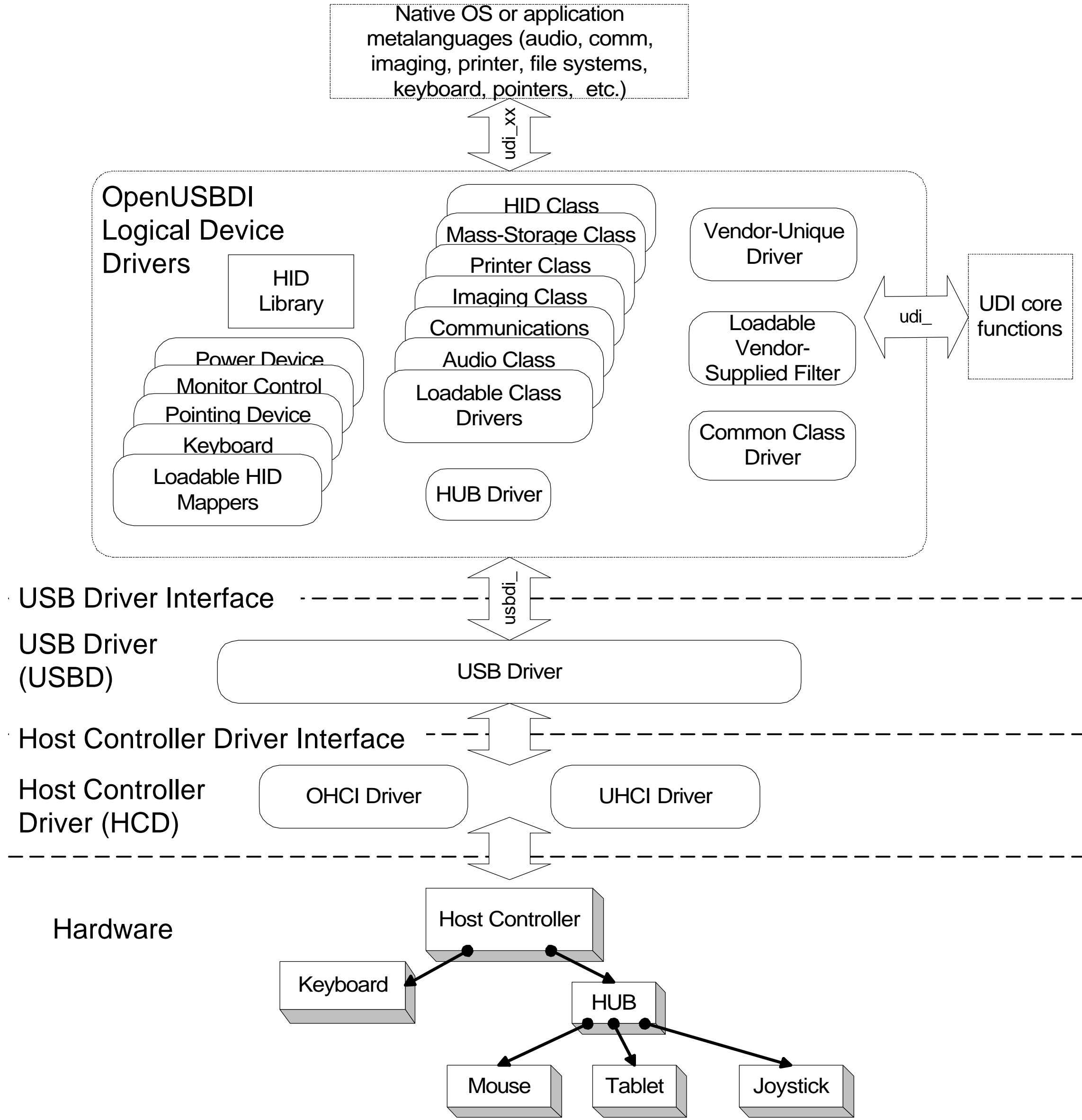

Figure 3 USB Environment

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

4 Functional Characteristics

4.1 Required Include Files

Before including any UDI header files, the LDD must 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 0x08:

```
#define OPEN_USBDI_VERSION 0x08
```

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

4.2 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[3]. 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.

4.3 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:

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

4.4 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.

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

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

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 5 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.

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

4.4.1 Logical-Device Driver Initialization

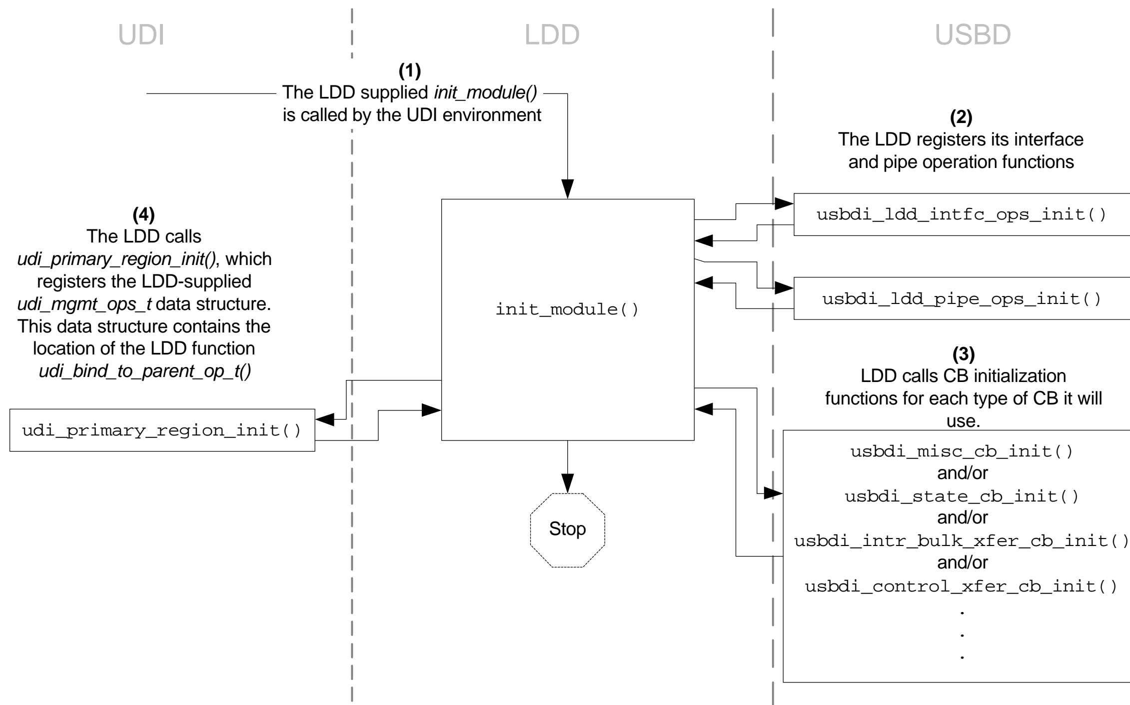

Figure 4: LDD Initialization

All LDDs shall provide an *init_module()* function. This function is called only once before any LDD instance is created and is responsible for:

1. setting up the LDD's scratch[4] space requirement for each type of OpenUSBDI control block (CB);

2. registering the LDD interface related operation functions and the pipe related operation functions with the USBD by calling *usbdi_ldd_intfc_ops_init()* and *usbdi_ldd_pipe_ops_init()*;

3. and initializing the LDD's primary region[5] by calling *udi_primary_region_init()*.

Figure 4 shows the functional flow of the LDD's *init_module()* function.

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

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

4.4.2 Interface Binding

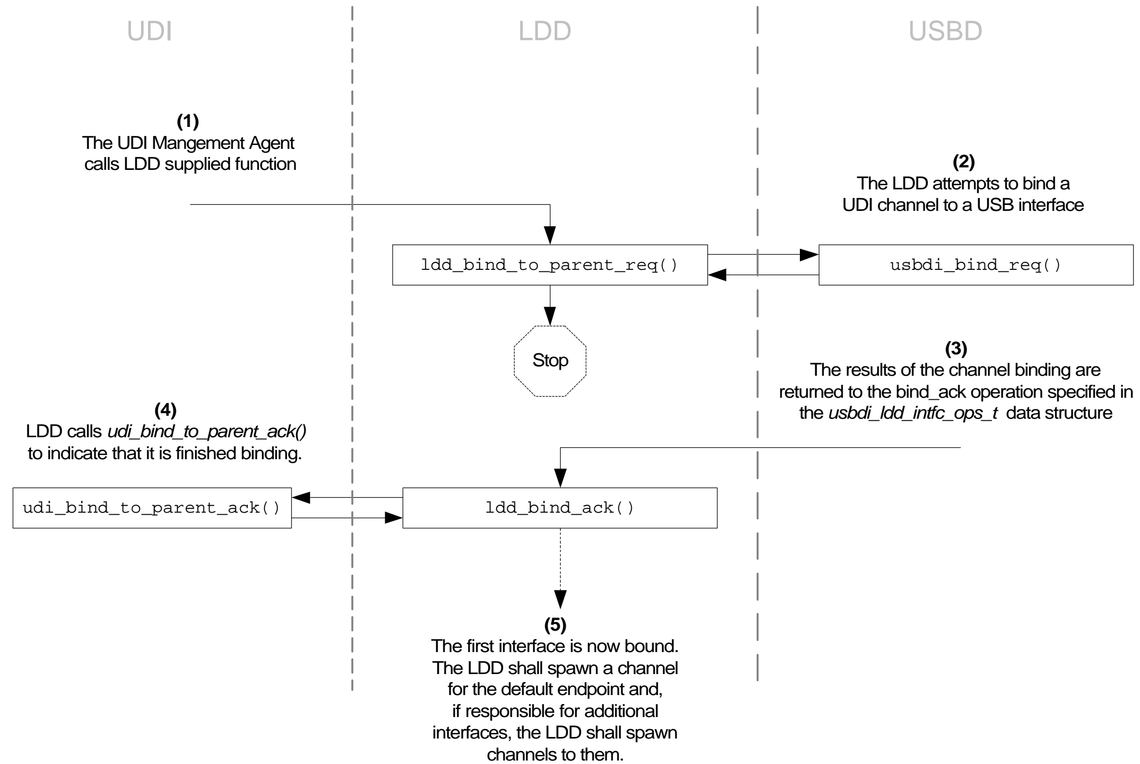

Figure 5: Interface Binding

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].

[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.

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

4.4.3 Setting the Configuration

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

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

4.4.4 Reading Descriptors

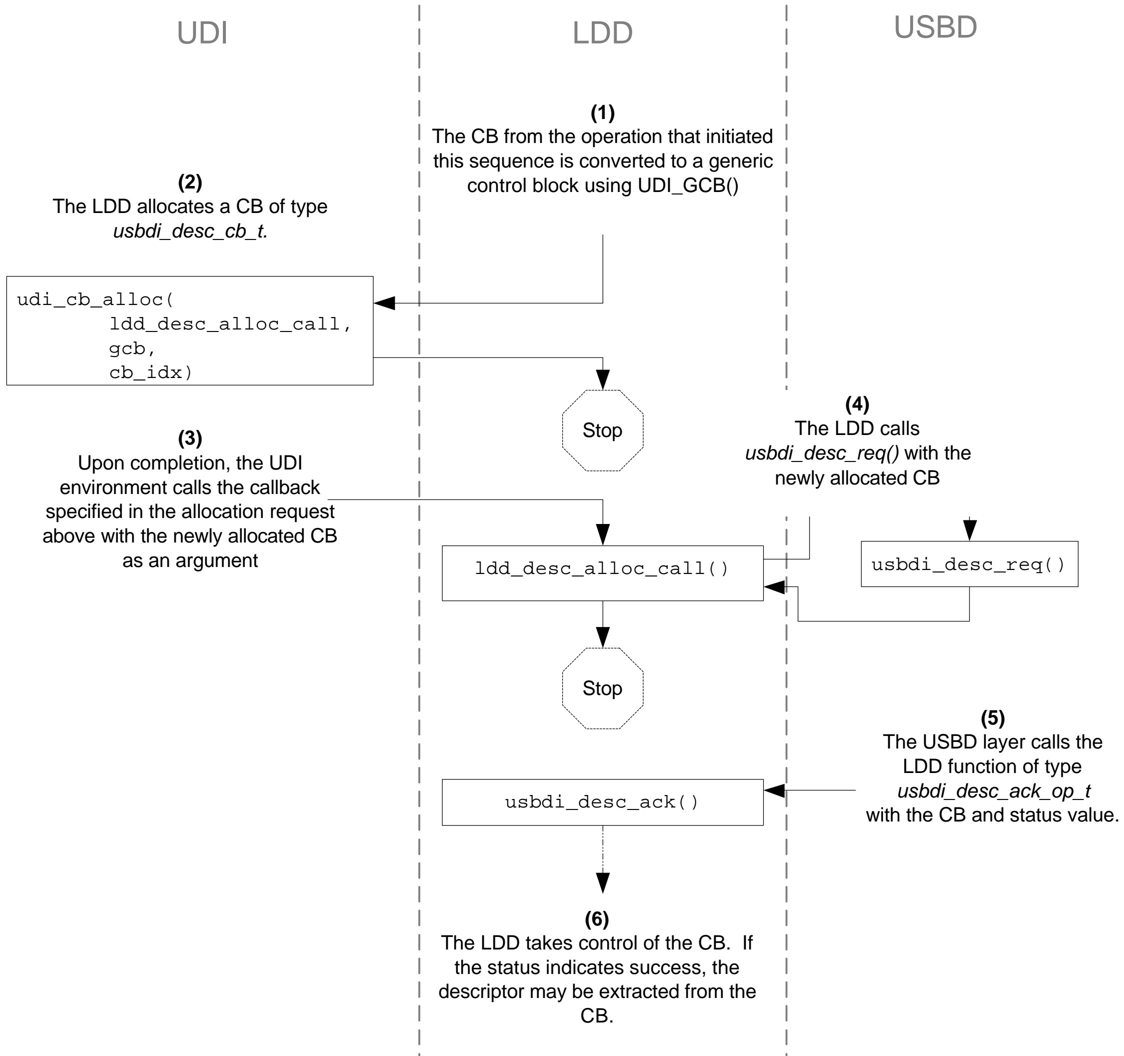

Figure 6: Reading Descriptors

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 6 shows the functional flow of a descriptor being retrieved.

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

4.4.5 Opening an Interface

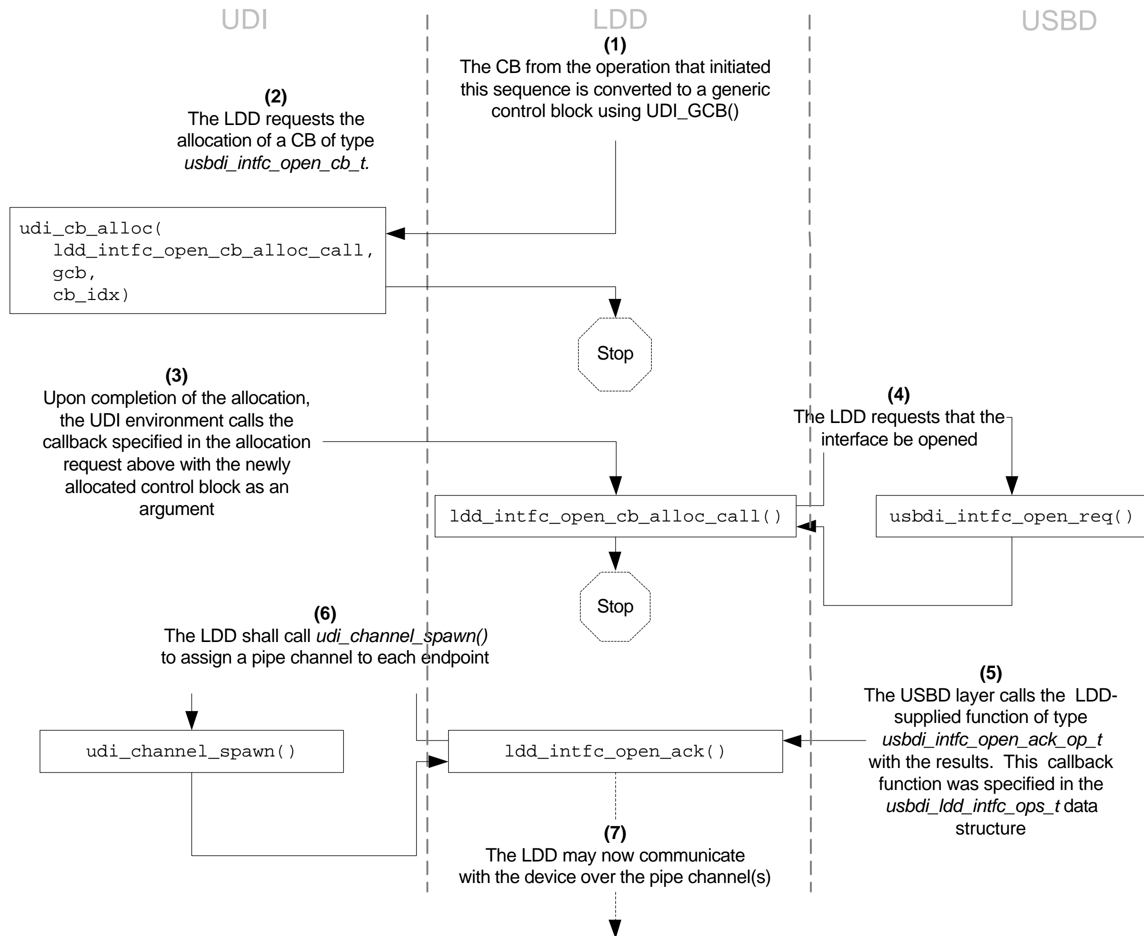

Figure 7: Opening an Interface

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 7 shows the functional flow of an interface being opened.

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

4.4.6 Closing an Interface

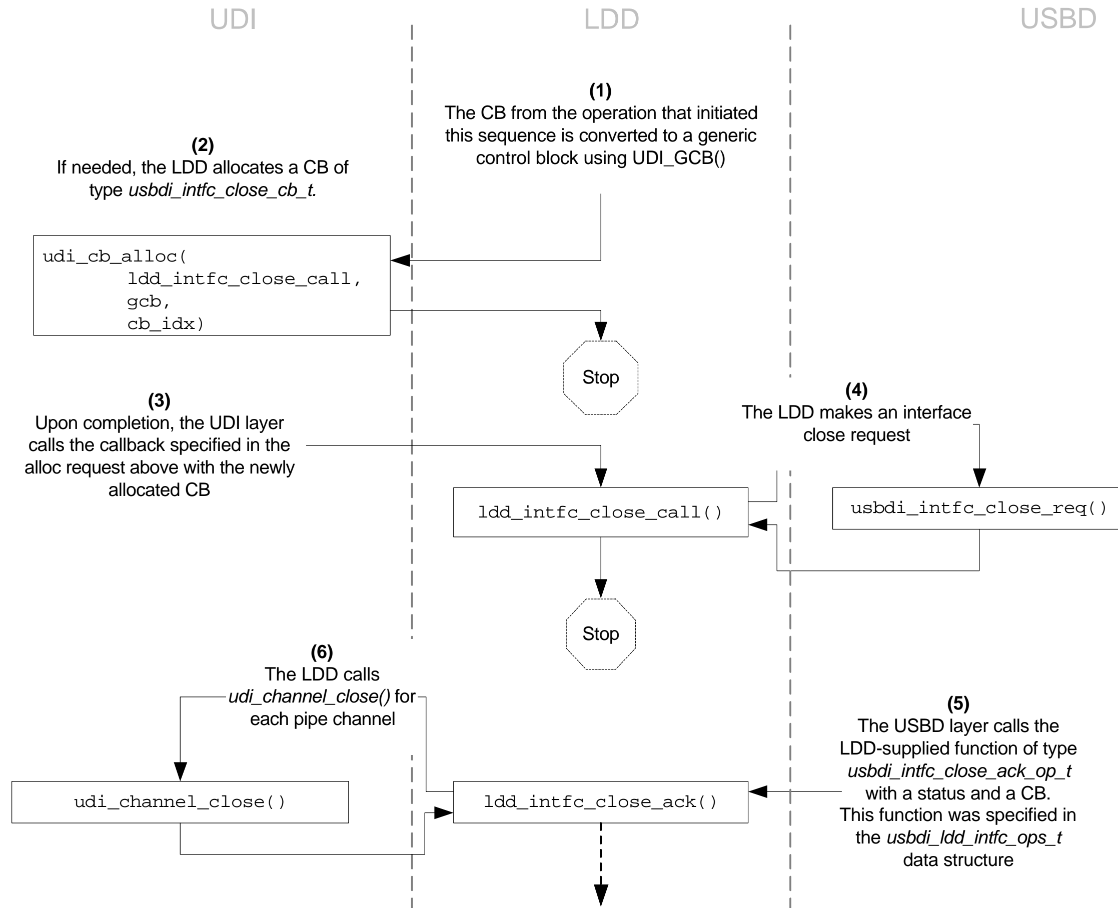

Figure 8: Closing an Interface

The LDD closes endpoint pipes (channels) by calling *udi_channel_close()* and closes interface channels by calling *usbdi_intfc_close_req()*. Figure 8 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_CLOSE type).

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

4.4.7 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 9 shows the code flow for a data transfer between the LDD and the USB device's endpoint.

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

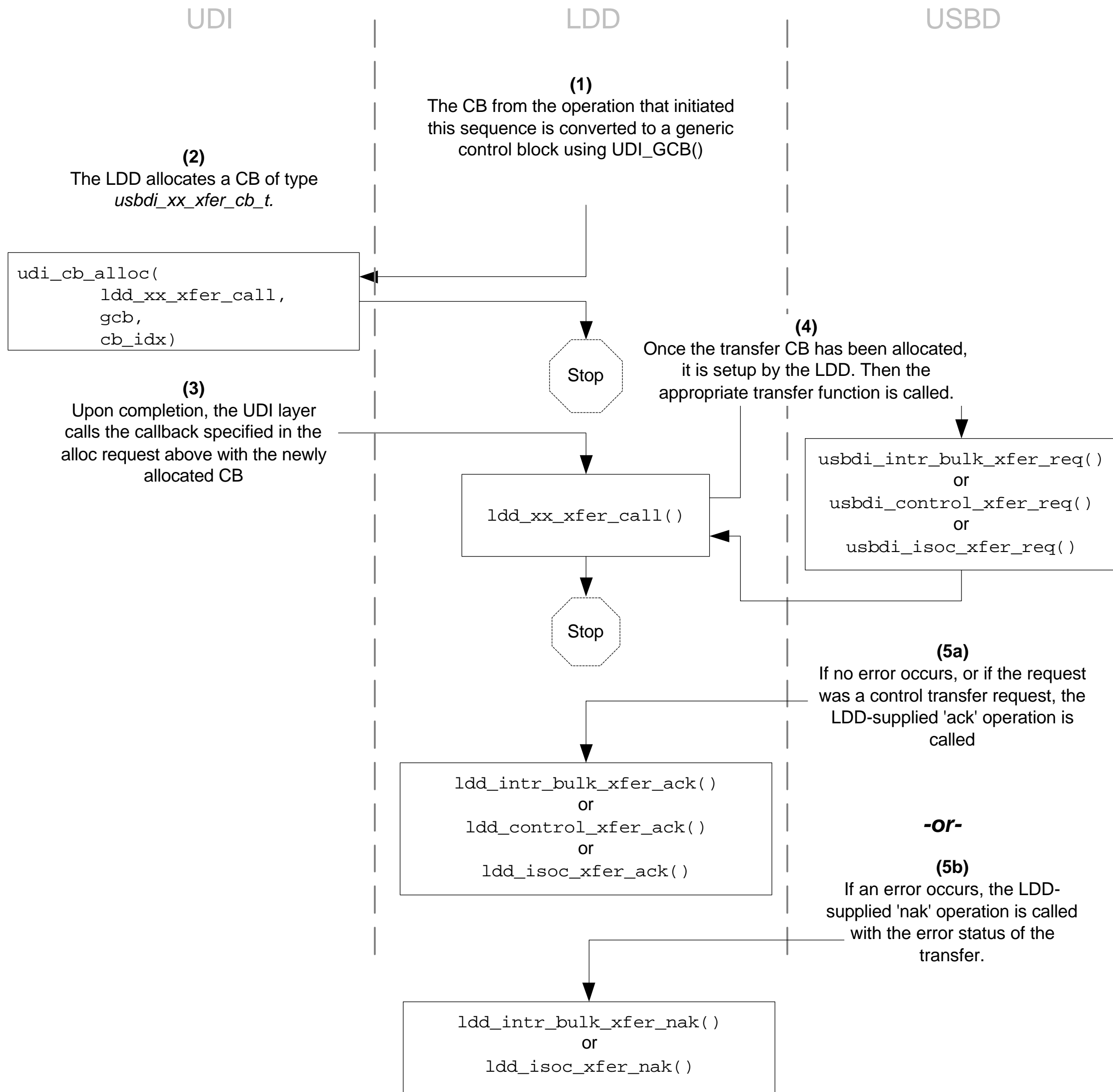

Figure 9: Transferring Data

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

4.4.8 Completing 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 4.4.1, "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.
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 *usbdi_xfer_abort_req()*, *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.

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

4.4.9 Pipe State Control

Pipe channels between the LDD and the device's endpoints are created by the LDD as described in Section 4.4.5, "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.
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.

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

4.4.10 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()*.

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

4.4.11 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.

4.4.12 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:

usbdi_xfer_abort_req()	The transfer control block identified by the *orig_tr_context* field of the *usbdi_xfer_abort_cb_t* 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.
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.

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

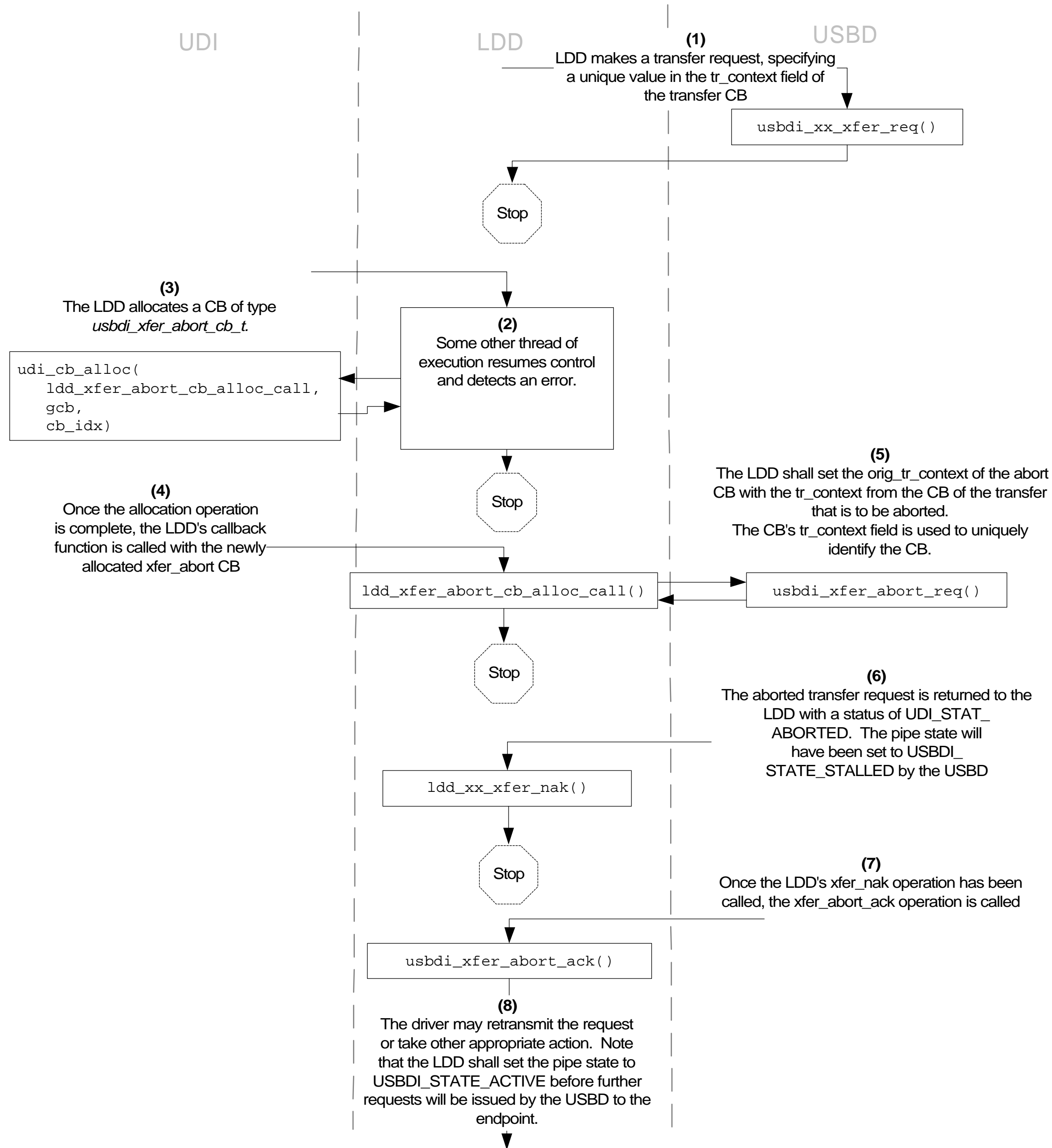

Figure 10: Aborting Transfers

Figure 10 shows the functional flow involved in aborting a transfer request during error recovery.

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

5 OpenUSBDI Metalanguage

The control block functions and function *typedef's* described in this section define how the LDD and the USBD communicate. Some 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_cb_init() — Called by the LDD, when the LDD is initialized (from *init_module()*). It sets up LDD specific values for the type of CB, such as the required scratch size.

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. This is the same index value that was given to *usbdi_XX_cb_init()* at driver initialization 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 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 interface may be derived by removing the first parameter of the callee-side entry point interface.

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

5.1 Driver Registration

All LDDs shall register their interface and pipe operation functions with the UDI Management Agent. These entry points are described by *usbdi_ldd_intfc_ops_t* and *usbdi_ldd_pipe_ops_t*. The LDD registers these entry points with the UDI Management Agent by calling *usbdi_ldd_intfc_ops_init()* and *usbdi_ldd_pipe_ops_init()*[9].

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

- *usbdi_ldd_intfc_ops_t*
- *usbdi_ldd_intfc_ops_init()*
- *usbdi_ldd_pipe_ops_t*
- *usbdi_ldd_pipe_ops_init()*

5.1.1 usbdi_ldd_intfc_ops_t

```
/*
 * Channel operation entry points 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_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;
```

[9] The equivalent USBD functions are not listed here but can be found in Appendix A.

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

5.1.2 usbdi_ldd_intfc_ops_init()

usbdi_ldd_intfc_ops_init() **– Called at least once by the LDD from *init_module()* to register the LDD interface operation entry points with the UDI Management Agent.**

5.1.2.1 Synopsis

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

typedef void
usbdi_ldd_intfc_ops_init(
                  udi_index_t ops_idx,
                  usbdi_ldd_intfc_ops_t *ops);
```

5.1.2.2 Arguments

udi_index_t ***ops_idx***	Index value that is to be associated with the interface related LDD function operations.
usbdi_ldd_intfc_ops_t ****ops***	Pointer to structure containing LDD interface operation functions.

5.1.2.3 Description

usbdi_ldd_intfc_ops_init() is called by the LDD's *init_module()* to register the LDD specific interface operation entry points. These entry points are described in the following sections: *usbdi_bind_ack_op_t (Section 5.3.2)*, *usbdi_unbind_ack_op_t (Section 5.4.2)*, *usbdi_intfc_open_ack_op_t (Section 5.5.2)*, *usbdi_intfc_close_ack_op_t (Section 5.6.2)*, *usbdi_frame_number_ack_op_t (Section 5.10.2)*, *usbdi_reset_device_ack_op_t (Section 5.11.2)*, *usbdi_intfc_abort_ack_op_t (Section 5.14.2)*, *usbdi_intfc_state_set_ack_op_t (Section 5.18.2)*, *usbdi_intfc_state_get_ack_op_t (Section 5.18.5)*, *usbdi_desc_ack_op_t (Section 5.20.4)*, *usbdi_device_state_get_ack_op_t (Section 5.19.2)*, *usbdi_config_set_ack_op_t (Section 5.21.2),* and *usbdi_async_event_ind_op_t (Section 5.22.1)*.

Operations that the LDD will never perform may be set to the corresponding "unused" function. For example, an LDD that will never perform a *usbdi_reset_device_req()* operation may set the *usbdi_reset_device_ack_op_t* field of *usbdi_ldd_intfc_ops_t* to *usbdi_reset_device_ack_unused()*. See the operation specific sections of this specification for details on "unused" functions.

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

5.1.3 usbdi_ldd_pipe_ops_t

```
/*
 * Channel operation entry points 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_xfer_abort_ack_op_t     *usbdi_xfer_abort_ack_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;
```


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

5.1.4 usbdi_ldd_pipe_ops_init()

usbdi_ldd_pipe_ops_init() **– Called at least once by the LDD from *init_module()* to register the LDD pipe operation functions with the UDI Management Agent.**

5.1.4.1 Synopsis

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

typedef void
usbdi_ldd_pipe_ops_init(
                  udi_index_t ops_idx,
                  usbdi_ldd_pipe_ops_t *ops);
```

5.1.4.2 Arguments

udi_index_t ***ops_idx***	Index value that is to be associated with the pipe related LDD function operations.
usbdi_ldd_pipe_ops_t ****ops***	Pointer to structure containing LDD pipe operation functions.

5.1.4.3 Description

usbdi_ldd_pipe_ops_init() is called by the LDD's *init_module()* to register the LDD specific pipe operation functions. The individual pipe function operation *typedefs* are described in the following sections: *usbdi_intr_bulk_xfer_ack_op_t* (Section 5.7.4), *usbdi_bulk_xfer_nak_op_t* (Section 5.7.5), *usbdi_control_xfer_ack_op_t* (Section 5.8.4), *usbdi_isoc_xfer_ack_op_t* (Section 5.9.6), *usbdi_isoc_xfer_nak_op_t* (Section 5.9.7), *usbdi_xfer_abort_ack_op_t* (Section 5.12.4), *usbdi_pipe_abort_ack_op_t* (Section 5.13.2), *usbdi_pipe_state_set_ack_op_t* (Section 5.16.2), *usbdi_pipe_state_get_ack_op_t* (Section 5.16.4), *usbdi_edpt_state_set_ack_op_t* (Section 5.17.2), and *usbdi_edpt_state_get_ack_op_t* (Section 5.17.4).

Operations that the LDD will never perform may be set to the corresponding operation specific "unused" function. For example, an LDD that will never perform a *usbdi_isoc_xfer_req()* operation may set the *usbdi_isoc_xfer_ack_op_t* field of *usbdi_ldd_pipe_ops_t* to *usbdi_isoc_xfer_ack_unsued()*. See the operation specific sections of this specification for details on "unused" functions.

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

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*
- *usbdi_misc_cb_init()*

5.2.1 usbdi_misc_cb_t

```
typedef struct {

      udi_cb_t gcb;

} usbdi_misc_cb_t;
```


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

5.2.2 usbdi_misc_cb_init()

***usbdi_misc_cb_init()* – Called at least once by the LDD from *init_module()* to set the requirements for the *usbdi_misc_cb_t*.**

5.2.2.1 Synopsis

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

void usbdi_misc_cb_init(
            udi_index_t cb_idx,
            udi_size_t scratch_requirement );
```

5.2.2.2 Arguments

udi_index_t ***cb_idx***	Index value to be associated with the *usbdi_misc_cb_t*.
udi_size_t ***scratch_requirement***	Scratch space size (in bytes) for the *usbdi_misc_cb_t* that the LDD associates with ***cb_idx***.

5.2.2.3 Description

The function *usbdi_misc_cb_init()* is called at least once by the LDD from *init_module()*. There is no callback or return value. The result is:

- The association by the Management Agent of the given ***cb_idx*** with the *usbdi_misc_cb_t*. This allows the LDD to allocate the *usbdi_misc_cb_t* by calling *udi_cb_alloc()* with an index value of ***cb_idx***.

- The scratch space size of the *usbdi_misc_cb_t* associated with ***cb_idx*** will be set to ***scratch_requirement***.

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

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 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_bind_req_op_t*
- *usbdi_bind_ack()*

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

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(
            udi_channel_t bind_channel,
            usbdi_misc_cb_t *cb);
```

5.3.1.2 Arguments

udi_channel_t ***bind_channel***	Channel of interface being bound.
usbdi_misc_cb_t ****cb***	Pointer to control block that was allocated with *udi_cb_alloc().*

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.

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

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 containing data 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, 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 are 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.

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

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 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_unbind_req_op_t*
- *usbdi_unbind_ack()*

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

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(
            udi_channel_t bind_channel,
            usbdi_misc_cb_t *cb);
```

5.4.1.2 Arguments

udi_channel_t ***bind_channel***	Channel of logical-device being unbound.
usbdi_misc_cb_t ****cb***	Pointer to control block that was allocated with *udi_cb_alloc().*

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).

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

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 containing data 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).

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

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 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_open_req_op_t*
- *usbdi_intfc_open_ack()*

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

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(
            udi_channel_t intfc_channel,
            usbdi_misc_cb_t *cb,
            udi_ubit8_t alternate_intfc,
            udi_ubit8_t open_flag );
```

5.5.1.2 Arguments

udi_channel ***intfc_channel***	UDI channel for USB interface that is being opened.
usbdi_misc_cb_t ****cb***	Pointer to control block that was allocate by *udi_cb_alloc()* with the *cb_idx* value that was given to *usbdi_misc_cb_init()*.
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 5.21.3 “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.

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

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 containing data 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.

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

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()*[10]. 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.

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

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

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()*

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

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(
            udi_channel_t intfc_channel,
            usbdi_misc_cb_t *cb );
```

5.6.1.2 Arguments

udi_channel_t ***intfc_channel***	Channel of interface being closed.
usbdi_misc_cb_t ****cb***	Pointer to control block that was allocated by *udi_cb_alloc()* with the *cb_idx* value that was given to *usbdi_misc_cb_init().*

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_CLOSE) for the pipe channels. The LDD is responsible for closing its end of the pipe channels after its *usbdi_intfc_close_ack_op_t* is called.

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.

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

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 containing data 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.

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

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_cb_init()*
- *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 only used by full speed USB devices. See the Universal Serial Bus Specification for more information.

5.7.1 usbdi_intr_bulk_xfer_cb_t

```
typedef struct {

      udi_cb_t gcb;          /* UDI generic control block */

      /*
       * Context pointer for this transfer request.  Used by
       * usbdi_xfer_abort_cb_t to uniquely identify the control block.
       */
      void *tr_context;

      /*
       * data_buf and data_len
       *
       * data_buf is set to refer to the buffer where the data
```


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

```
         * will be transferred to/from.  The LDD sets the data_len field
         * to the maximum number of bytes that may be transferred.  To
         * specify a zero length data transfer the data_len shall be set
         * by the LDD to zero and data_buf will be set to NULL_BUF.
         * If the data_len is non-zero, the
         * data_buf shall be set to a valid buffer handle.
         *
         * On completion, the USBD sets the data_len field with the
         * actual number of bytes transferred.
         */
        udi_buf_t data_buf;
        udi_size_t data_len;

        /*
         * 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 only valid flag for
         * interrupt and bulk requests is USBDI_XFER_SHORT_OK that
         * specifies a final transfer count that is less than data_len
         * will NOT generate an error and will NOT stall the
         * pipe.  Note that the direction of the data is implied in the
         * direction of the endpoint.
         */
        udi_ubit8_t xfer_flags;
#define USBDI_XFER_SHORT_OK (1<<0)

} usbdi_intr_bulk_xfer_cb_t;
```


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

5.7.2 usbdi_intr_bulk_xfer_cb_init()

***usbdi_intr_bulk_xfer_cb_init()* – Called at least once by the LDD from *init_module()* to set the requirements for *usbdi_intr_bulk_xfer_cb_t* if the LDD is going to use the *usbdi_intr_bulk_xfer_cb_t*.**

5.7.2.1 Synopsis

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

void usbdi_intr_bulk_xfer_cb_init(
            udi_index_t cb_idx,
            udi_size_t scratch_requirement );
```

5.7.2.2 Arguments

udi_index_t ***cb_idx***	Index value to be associated with the *usbdi_intr_bulk_xfer_cb_t*.
udi_size_t ***scratch_requirement***	Scratch space size (in bytes) needed by the LDD for the *usbdi_intr_bulk_xfer_cb_t*.

5.7.2.3 Description

The function *usbdi_intr_bulk_xfer_cb_init()* is called at least once by the LDD from *init_module()* if the LDD is going to use the *usbdi_intr_bulk_xfer_cb_t*. There is no callback or return value. The result is:

- The association by the Management Agent of the given ***cb_idx*** with the *usbdi_intr_bulk_xfer_cb_t*. This allows the LDD to allocate the *usbdi_intr_bulk_xfer_cb_t* by calling *udi_cb_alloc()* with an index value of ***cb_idx***.

- The scratch space size for the *usbdi_intr_bulk_xfer_cb_t* will be set to ***scratch_requirement***.

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

5.7.3 usbdi_intr_bulk_xfer_req()

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

5.7.3.1 Synopsis

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

void usbdi_intr_bulk_xfer_req(
            udi_channel_t pipe_channel,
            usbdi_intr_bulk_xfer_cb_t *cb );
```

5.7.3.2 Arguments

udi_channel_t ***pipe_channel***	Channel for the pipe connecting the LDD with the interrupt or bulk endpoint.
usbdi_intr_bulk_xfer_cb_t ****cb***	Pointer to control block that was allocated by *udi_cb_alloc()* with the *cb_idx* value that was given to *usbdi_intr_bulk_xfer_cb_init().*

5.7.3.3 Description

usbdi_intr_bulk_xfer_req() is called by the LDD to transfer data with an interrupt or bulk endpoint. The pipe described by ***pipe_channel*** shall have been spawned by the LDD with a call to *udi_channel_spawn()*. The direction of the transfer is inherent in the type of endpoint.

The *data_len* 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. If *data_len* is zero, *data_buf* shall be set to NULL_BUF. For output, *data_buf* shall contain *data_len* 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. (See Section 5.11.3, “Aborting a Transfer”.)

The *xfer_flags* field may be set to zero (“0”) or USBDI_XFER_SHORT_OK. If USBDI_XFER_SHORT_OK is set and the device transfers less than *data_len* bytes, the request will complete without an error and the LDD’s *usbdi_intr_bulk_xfer_ack_op_t* function will be called. See Section 5.7.4, *usbdi_intr_bulk_xfer_ack_op_t,* for more information.

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

The *tr_context* field of the CB shall be set by the LDD to a value that is unique and not duplicated by CBs currently outstanding or queued to ***pipe_channel***. This unique value may be a pointer to a structure private to the LDD that is specific to the request. The *tr_context* value is used by *usbdi_xfer_abort_req()* to identify the transfer control block being aborted.

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 5.16.3, *usbdi_pipe_state_get_req(),* for more information.

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

5.7.4 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.**

5.7.4.1 Synopsis

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

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

5.7.4.2 Arguments

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

5.7.4.3 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_len* shall be examined by the LDD to determine the actual number of bytes transferred.

If the direction of the data flow of the pipe channel is from the LDD to the device, the *data_buf* of CB shall be set to NULL_BUF by the USBD and the buffer shall be freed or reused by the USBD. The LDD shall no longer use the buffer that had been used to set up the *data_buf* field of *usbdi_intr_bulk_xfer_cb_t.*

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().*

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

5.7.5 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.**

5.7.5.1 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 );
```

5.7.5.2 Arguments

usbdi_intr_bulk_xfer_cb_t ****cb***	Pointer to control block containing data 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 *usbdi_xfer_abort_req()*, *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.

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

UDI_STAT_MISTAKEN_IDENTITY	A field in the CB is invalid.

5.7.5.3 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. See Section 5.16, "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()*.

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

5.7.6 usbdi_intr_bulk_xfer_ack_unused()

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

5.7.6.1 Synopsis

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

usbdi_intr_bulk_xfer_ack_op_t usbdi_intr_bulk_xfer_ack_unused;
```

5.7.6.2 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().*

5.7.7 usbdi_intr_bulk_xfer_nak_unused()

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

5.7.7.1 Synopsis

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

usbdi_intr_bulk_xfer_nak_op_t usbdi_intr_bulk_xfer_nak_unused;
```

5.7.7.2 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().*

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

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_cb_init()*
- *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

```
typedef struct {

      udi_cb_t gcb;     /* UDI generic control block */

      /*
       * Context pointer for this transfer request.  Used by
       * usbdi_xfer_abort_cb_t to uniquely identify the control block.
       */
      void *tr_context;

      /*
       * 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;
```


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

```
        /*
         * 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 that 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 and data_buf shall be set to
         * NULL_BUF.  If the data_len is non-zero, data_buf shall contain a
         * valid handle.
         *
         * On completion the USBD sets the data_len field with the actual
         * number of bytes transferred.
         */
        udi_buf_t data_buf;
        udi_size_t data_len;

        /*
         * 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).
         * USBDI_XFER_SHORT_OK (see usbdi_intr_bulk_xfer_cb_t) is also
         * a valid flag.
         */
        udi_ubit8_t xfer_flags;
#define USBDI_XFER_IN  (1<<2) /* transfer data from the device */
#define USBDI_XFER_OUT (1<<3) /* transfer data to the device */

} usbdi_control_xfer_cb_t;
```


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

5.8.2 usbdi_control_xfer_cb_init()

usbdi_control_xfer_cb_init() **– Called at least once by the LDD from** ***init_module()*** **to set the requirements for the** ***usbdi_control_xfer_cb_t*** **if the LDD is going to use a** ***usbdi_control_xfer_cb_t*****.**

5.8.2.1 Synopsis

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

void usbdi_control_xfer_cb_init(
            udi_index_t cb_idx,
            udi_size_t scratch_requirement );
```

5.8.2.2 Arguments

udi_index_t ***cb_idx***	Index value to be associated with the *usbdi_control_xfer_cb_t*.
udi_size_t ***scratch_requirement***	Scratch space size (in bytes) needed by the LDD for the *usbdi_control_xfer_cb_t*.

5.8.2.3 Description

The function *usbdi_control_xfer_cb_init()* is called at least once by the LDD from *init_module()* if the LDD uses a *usbdi_control_xfer_cb_t*. There is no callback or return value. The result is:

- The Management Agent associates the given ***cb_idx*** with the *usbdi_control_xfer_cb_t*. This allows the LDD to allocate the *usbdi_control_xfer_cb_t* by calling *udi_cb_alloc()* with an index value of ***cb_idx***.

- The scratch space size for the *usbdi_control_xfer_cb_t* will be set to ***scratch_requirement***. Once a *usbdi_control_xfer_cb_t* is allocated, a buffer of size ***scratch_requirement*** number of bytes is available for use by the LDD. This buffer is located in *gcb.scratch* of *usbdi_control_xfer_cb_t*.

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

5.8.3 usbdi_control_xfer_req()

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

5.8.3.1 Synopsis

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

void usbdi_control_xfer_req(
            udi_channel_t pipe_channel,
            usbdi_control_xfer_cb_t *cb );
```

5.8.3.2 Arguments

udi_channel_t ***pipe_channel***	Channel for the pipe connecting the LDD with the control endpoint.
usbdi_control_xfer_cb_t ****cb***	Pointer to control block that was allocated by *udi_cb_alloc()* with the *cb_idx* value that was given to *usbdi_control_xfer_cb_init().*

5.8.3.3 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_len* 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_len* is zero (“0”), *data_buf* shall be set by the LDD to NULL_BUF. If *data_len* is greater than zero, *data_buf* shall point to a buffer that is large enough to accommodate *data_len* 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.

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

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_len* 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_len* bytes, the control block will complete with a status of UDI_OK. See Section 5.8.4, *usbdi_control_xfer_ack_op_t,* for more information.

The *tr_context* field of the CB shall be set by the LDD to a value that is unique and not duplicated by other CBs that the LDD queues to ***pipe_channel***. This unique value may be a pointer to a structure private to the LDD that is specific to the request. The *tr_context* value is used by *usbdi_xfer_abort_req()* to identify the transfer control block being aborted.

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 5.16.3, *usbdi_pipe_state_get_req(),* for more information.

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

5.8.4 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.**

5.8.4.1 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 );
```

5.8.4.2 Arguments

usbdi_control_xfer_cb_t ****cb***	Pointer to the control block containing data 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 *usbdi_xfer_abort_req()*, *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.

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

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.

5.8.4.3 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_len* shall be examined to determine the actual number of bytes transferred. In all cases, *data_len* will contain the actual number of bytes transferred during the data phase.

It an error occurred the state of the pipe shall be set by the USBD to USBDI_STATE_STALLED before this function is called. See Section 5.16, "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()*.

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

5.8.5 usbdi_control_xfer_ack_unused()

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

5.8.5.1 Synopsis

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

usbdi_control_xfer_ack_op_t usbdi_control_xfer_ack_unused;
```

5.8.5.2 Description

usbdi_control_xfer_ack_unused() maybe 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()*.

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

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_cb_init()*
- *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()*

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

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. The number of *usbdi_isoc_frame_request_t* structures contained by this array is specified by the LDD at driver initialization time when it calls *usbdi_isoc_xfer_cb_init().*

The *usbdi_isoc_frame_request_t* contains two fields, one describing the length of the data to be transferred, and one that will receive the status of the transfer. Figure 11 shows this arrangement.

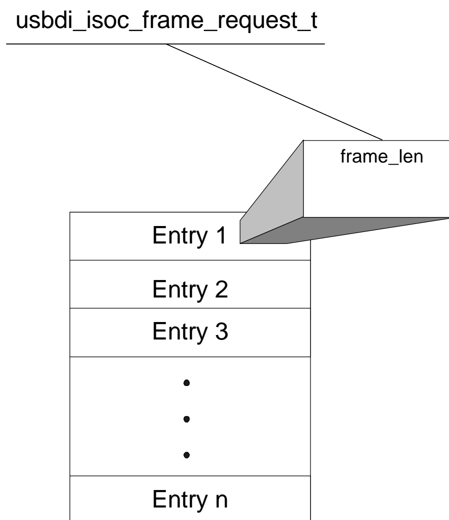

Figure 11

5.9.2 usbdi_isoc_frame_request_t

```
typedef struct {
      udi_size_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;
```


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

5.9.3 usbdi_isoc_xfer_cb_t

```
typedef struct {

      udi_cb_t gcb;     /* UDI generic control block */

      /*
       * Context pointer for this transfer request.  Used by
       * usbdi_xfer_abort_cb_t to uniquely identify the control block.
       */
      void *tr_context;

      /*
       * 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 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 this flag is set, the frame_number field will
       * be ignored by the USBD.  The direction of the data is not specified
       * since the direction is implied by the direction of the endpoint.
       * USBDI_XFER_SHORT_OK (see usbdi_intr_bulk_xfer_cb_t) is also a
       * valid flag.
       */
      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.
```


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

```
        */
      udi_ubit32_t frame_number;
} usbdi_isoc_xfer_cb_t;
```


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

5.9.4 usbdi_isoc_xfer_cb_init()

usbdi_isoc_xfer_cb_init() **– Called at least once by the LDD from *init_module()* to set up the requirements for *usbdi_isoc_xfer_cb_t* if the LDD uses a *usbdi_isoc_xfer_cb_t.***

5.9.4.1 Synopsis

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

void usbdi_isoc_xfer_cb_init(
      udi_index_t cb_idx,
      udi_size_t scratch_requirement,
      udi_ubit8_t n_frame_elem );
```

5.9.4.2 Arguments

udi_index_t ***cb_idx***	Index value to be associated with the *usbdi_isoc_xfer_cb_t.*
udi_size_t ***scratch_requirement***	Scratch space size (in bytes) needed by the LDD for the *usbdi_isoc_xfer_cb_t* .
udi_ubit8_t ***n_frame_elem***	Number of frames that will be available in the *frame_array* field of the *usbdi_isoc_xfer_cb_t.*

5.9.4.3 Description

The function *usbdi_isoc_xfer_cb_init()* is called at least once by the LDD from *init_module()* if the LDD uses a *usbdi_isoc_xfer_cb_t.* There is no callback or return value. The result is:

- The Management Agent associates the given ***cb_idx*** with the *usbdi_isoc_xfer_cb_t.* This allows the LDD to allocate the *usbdi_isoc_xfer_cb_t* by calling *udi_cb_alloc()* with an index value of ***cb_idx***.
- The scratch space size for the *usbdi_isoc_xfer_cb_t* will be set to ***scratch_requirement***.
- The number of elements in the *frame_array* of maximum number of *usbdi_isoc_xfer_cb_t* is set to ***n_frame_elem***.

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

5.9.5 usbdi_isoc_xfer_req()

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

5.9.5.1 Synopsis

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

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

5.9.5.2 Arguments

udi_channel_t ***pipe_channel***	Channel for the pipe connecting the LDD with the isochronous endpoint.
usbdi_isoc_xfer_cb_t ****cb***	Pointer to a control block that was allocated with *udi_cb_alloc()* with the *cb_idx* value that was given to *usbdi_isoc_xfer_cb_init()*.

5.9.5.3 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 direction of the transfer is inherent in the type of endpoint.

The *frame_array* of the CB points to an array of *usbdi_isoc_frame_request_t* elements. The number of elements in the array will be ***n_frame_elem*** (see Section 5.9.4, *usbdi_isoc_xfer_cb_init()*). The LDD may set up zero to ***n_frame_elem*** of these elements. 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 (see Section 5.9.6, *usbdi_isoc_xfer_ack_op_t,* for more information).

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_BUF 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 5.9.7, “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

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

USBDI_XFER_ASAP in the *xfer_flags* field. In this case, *frame_number* will not be examined by the USBD.

The *tr_context* field of the CB shall be set by the LDD to a value that is unique and not duplicated by other CBs that the LDD queues to ***pipe_channel***. This unique value may be a pointer to a structure private to the LDD that is specific to the request. The *tr_context* value is used by *usbdi_xfer_abort_req()* to identify the transfer control block being aborted.

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 5.16.3, *usbdi_pipe_state_get_req(),* for more information.

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

5.9.6 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.**

5.9.6.1 Synopsis

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

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

5.9.6.2 Arguments

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

5.9.6.3 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.

If the data flow is from the device to *data_buf* and USBDI_XFER_SHORT_OK was set by the LDD in the CB's *xfer_flags* field the LDD shall examine *frame_len* of each element in the *frame_array* to determine if, due to a short transfer, there are holes (missing or invalid data) in *data_buf.* If the direction of the data flow of the pipe channel is from the LDD to the device, the *data_buf* of CB shall be set to NULL_BUF by the USBD and the buffer shall be freed or reused by the USBD. The LDD shall no longer use the buffer that had been used to set up the *data_buf* field of *usbdi_isoc_xfer_cb_t.*

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

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

5.9.7 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.**

5.9.7.1 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 );
```

5.9.7.2 Arguments

usbdi_isoc_xfer_cb_t ****cb***	Pointer to control block containing data 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 *usbdi_xfer_abort_req()*, *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.

5.9.7.3 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 5.16, "Getting and Setting Pipe States", for more information.

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

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.

If the data flow is from the device to *data_buf*, the LDD shall examine *frame_len* of each element in the *frame_array* to determine if, due to a short transfer, there are holes (missing or invalid data) in *data_buf*. The *frame_status* shall also be examined to determine if the data received in each frame is valid.

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()*.

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

5.9.8 usbdi_isoc_xfer_ack_unused()

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

5.9.8.1 Synopsis

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

usbdi_isoc_xfer_ack_op_t usbdi_isoc_xfer_ack_unused;
```

5.9.8.2 Description

usbdi_isoc_xfer_ack_unused() maybe 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()*.

5.9.9 usbdi_isoc_xfer_nak_unused()

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

5.9.9.1 Synopsis

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

usbdi_isoc_xfer_nak_op_t usbdi_isoc_xfer_nak_unused;
```

5.9.9.2 Description

usbdi_isoc_xfer_nak_unused() maybe 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()*.

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

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()*

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

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(
            udi_channel_t intfc_channel,
            usbdi_misc_cb_t *cb );
```

5.10.1.2 Arguments

udi_channel_t ***intfc_channel***	Channel for the interface.
usbdi_misc_cb_t ****cb***	Pointer to a control block that was allocated with *udi_cb_alloc()* with the *cb_idx* value that was given to *usbdi_misc_cb_init().*

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 request does not generate any activity on the USB.

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

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 containing data 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.

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

5.10.3 usbdi_frame_number_ack_unused()

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

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

5.11 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()*

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

5.11.1 usbdi_reset_device_req()

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

5.11.1.1 Synopsis

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

void usbdi_reset_device_req(
            udi_channel_t intfc_channel,
            usbdi_misc_cb_t *cb );
```

5.11.1.2 Arguments

udi_channel_t ***intfc_channel***	Interface channel to which the device is bound to.
usbdi_misc_cb_t ****cb***	Pointer to control block that was allocated with *udi_cb_alloc()* with the *cb_idx* value that was given to *usbdi_misc_cb_init()*.

5.11.1.3 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.

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

5.11.2 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.**

5.11.2.1 Synopsis

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

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

5.11.2.2 Arguments

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

5.11.2.3 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.

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

5.11.3 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()*.**

5.11.3.1 Synopsis

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

usbdi_reset_device_ack_op_t usbdi_reset_device_ack_unused;
```

5.11.3.2 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()*.

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

5.12 Aborting a Transfer

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

- *usbdi_xfer_abort_cb_t*
- *usbdi_xfer_abort_cb_init()*
- *usbdi_xfer_abort_req()*
- *usbdi_xfer_abort_ack_op_t*
- *usbdi_xfer_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_xfer_abort_req_op_t*
- *usbdi_xfer_abort_ack()*

5.12.1 usbdi_xfer_abort_cb_t

```
typedef struct {

      udi_cb_t gcb;            /* UDI generic control block */
      void *orig_tr_context;   /* Context from the CB to be aborted */
      void *tr_context;        /* Context for the abort CB itself */

} usbdi_xfer_abort_cb_t;
```


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

5.12.2 usbdi_xfer_abort_cb_init()

***usbdi_xfer_abort_cb_init()* – Called at least once by the LDD from *init_module()* to set up the requirements of *usbdi_xfer_abort_cb_t* if the LDD uses the *usbdi_xfer_abort_cb_t*.**

5.12.2.1 Synopsis

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

void usbdi_xfer_abort_cb_init(
            udi_index_t cb_idx,
            udi_size_t scratch_requirement );
```

5.12.2.2 Arguments

udi_index_t ***cb_idx***	Index value to be associated with the *usbdi_xfer_abort_cb_t*.
udi_size_t ***scratch_requirement***	Scratch space size (in bytes) needed by the LDD for the *usbdi_xfer_abort_cb_t*.

5.12.2.3 Description

The function *usbdi_xfer_abort_cb_init()* is called at least once by the LDD from *init_module()* if the LDD uses a *usbdi_xfer_abort_cb_t*. There is no callback or return value. The result is:

- The Management Agent associates the given ***cb_idx*** with the *usbdi_xfer_abort_cb_t*. This allows the LDD to allocate the *usbdi_xfer_abort_cb_t* by calling *udi_cb_alloc()* with an index value of ***cb_idx***.
- The scratch space size for the *usbdi_xfer_abort_cb_t* will be set to ***scratch_requirement***.

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

5.12.3 usbdi_xfer_abort_req()

***usbdi_xfer_abort_req()* – Called by the LDD to abort a transfer control block.**

5.12.3.1 Synopsis

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

void usbdi_xfer_abort_req(
            udi_channel_t pipe_channel,
            usbdi_xfer_abort_cb_t *cb );
```

5.12.3.2 Arguments

udi_channel_t ***pipe_channel***	Channel for the pipe to which the transfer control block that is being aborted was queued.
usbdi_xfer_abort_cb_t ****cb***	Pointer to a control block that was allocated with *udi_cb_alloc()* with the *cb_idx* value that was given to *usbdi_xfer_abort_cb_init().*

5.12.3.3 Description

usbdi_xfer_abort_req() is called by the LDD to abort a control block that was issued to ***pipe_channel*** with a call to *usbdi_intr_bulk_xfer_req()*, *usbdi_control_xfer_req()*, or *usbdi_isoc_xfer_req().*

The *orig_tr_context* of the CB shall be set to the value that had been stored in *tr_context* of the transfer control block that is to be aborted.

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

5.12.4 usbdi_xfer_abort_ack_op_t

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

5.12.4.1 Synopsis

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

typedef
void usbdi_xfer_abort_ack_op_t(
            usbdi_xfer_abort_cb_t *cb,
            udi_status_t status );
```

5.12.4.2 Arguments

usbdi_xfer_abort_cb_t ****cb***	Pointer to a control block containing data that was passed to *usbdi_xfer_abort_req().*

5.12.4.3 Description

The LDD shall supply a *usbdi_xfer_abort_ack_op_t* function in its *usbdi_ldd_pipe_ops_t*. This function is called in response to the LDD's calling *usbdi_xfer_abort_req()*. If the abort was successful, the aborted transfer control block will be returned to the appropriate *xfer_ack_op_t* (for control endpoints) or *xfer_nak_op_t* LDD function with a status of UDI_STAT_ABORTED. The *usbdi_xfer_abort_cb_t* is then returned to the LDD's *usbdi_xfer_abort_ack_op_t* function.

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

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

5.12.5 usbdi_xfer_abort_ack_unused()

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

5.12.5.1 Synopsis

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

usbdi_xfer_abort_ack_op_t usbdi_xfer_abort_ack_unused;
```

5.12.5.2 Description

usbdi_xfer_abort_ack_unused() maybe used in the *usbdi_ldd_pipe_ops_t* as the *usbdi_xfer_abort_ack_op* if the LDD will never call *usbdi_xfer_abort_req()*.

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

5.13 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 5.16, "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()*

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

5.13.1 usbdi_pipe_abort_req()

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

5.13.1.1 Synopsis

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

void usbdi_pipe_abort_req(
            udi_channel_t pipe_channel,
            usbdi_misc_cb_t *cb );
```

5.13.1.2 Arguments

udi_channel_t ***pipe_channel***	Channel for pipe that is to be aborted.
usbdi_misc_cb_t ****cb***	Pointer to a control block that was allocated with *udi_cb_alloc()* with the *cb_idx* value that was given to *usbdi_misc_cb_init()*.

5.13.1.3 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.

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

5.13.2 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.**

5.13.2.1 Synopsis

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

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

5.13.2.2 Arguments

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

5.13.2.3 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.

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

5.13.3 usbdi_pipe_abort_ack_unused()

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

5.13.3.1 Synopsis

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

usbdi_pipe_abort_ack_op_t usbdi_pipe_abort_ack_unused;
```

5.13.3.2 Description

usbdi_pipe_abort_ack_unused() maybe 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()*.

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

5.14 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()*

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

5.14.1 usbdi_intfc_abort_req()

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

5.14.1.1 Synopsis

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

void usbdi_intfc_abort_req(
            udi_channel_t intfc_channel,
            usbdi_misc_cb_t *cb );
```

5.14.1.2 Arguments

udi_channel_t ***intfc_channel***	Channel to the USB interface that is to be aborted.
usbdi_misc_cb_t ****cb***	Pointer to a control block that was allocated with *udi_cb_alloc()* with the *cb_idx* value that was given to *usbdi_misc_cb_init()*.

5.14.1.3 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.

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

5.14.2 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.**

5.14.2.1 Synopsis

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

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

5.14.2.2 Arguments

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

5.14.2.3 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* 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 5.17.5, "Getting and Setting Interface States", for more information.

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

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

5.14.3 usbdi_intfc_abort_ack_unused()

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

5.14.3.1 Synopsis

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

usbdi_intfc_abort_ack_op_t usbdi_intfc_abort_ack_unused;
```

5.14.3.2 Description

usbdi_intfc_abort_ack_unused() maybe 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()*.

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

5.15 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.

5.15.1 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;
```


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

5.15.2 usbdi_state_cb_init()

usbdi_state_cb_init() **– Called at least once by the LDD from *init_module()* to set up the requirements for *usbdi_state_cb_t*.**

5.15.2.1 Synopsis

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

void usbdi_state_cb_init(
            udi_index_t cb_idx,
            udi_size_t scratch_requirement );
```

5.15.2.2 Arguments

udi_index_t ***cb_idx***	Index value to be associated with the *usbdi_state_cb_t*.
udi_size_t ***scratch_requirement***	Scratch space size (in bytes) needed by the LDD for the *usbdi_state_cb_t*.

5.15.2.3 Description

The function *usbdi_state_cb_init()* is called at least once by the LDD from *init_module()*. There is no callback or return value. The result is:

- The Management Agent associates the given ***cb_idx*** with the *usbdi_state_cb_t*. This allows the LDD to allocate the *usbdi_state_cb_t* by calling *udi_cb_alloc()* with an index value of ***cb_idx***.

The scratch space size for the *usbdi_state_cb_t* will be set to ***scratch_requirement***.

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

5.16 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()*

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

5.16.1 usbdi_pipe_state_set_req()

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

5.16.1.1 Synopsis

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

void usbdi_pipe_state_set_req(
            udi_channel_t pipe_channel,
            usbdi_state_cb_t *cb );
```

5.16.1.2 Arguments

udi_channel_t ***pipe_channel***	Channel for the pipe.
usbdi_state_cb_t ****cb***	Pointer to a control block that was allocated with *udi_cb_alloc()* with the *cb_idx* value that was given to *usbdi_state_cb_init()*.

5.16.1.3 Description

usbdi_pipe_state_set_req() is called by the LDD to change the state of the pipe described by ***pipe_channel***. 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 5.17.5, “Getting and Setting Interface States”, for more information.

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

5.16.2 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.**

5.16.2.1 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 );
```

5.16.2.2 Arguments

usbdi_state_cb_t ***cb***	Pointer to a control block containing data 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.

5.16.2.3 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.

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

5.16.3 usbdi_pipe_state_get_req()

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

5.16.3.1 Synopsis

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

void usbdi_pipe_state_get_req(
            udi_channel_t pipe_channel,
            usbdi_state_cb_t *cb);
```

5.16.3.2 Arguments

udi_channel_t ***pipe_channel***	Channel for the pipe.
usbdi_state_cb_t ****cb***	Pointer to a control block that was allocated with *udi_cb_alloc()* with the *cb_idx* value that was given to *usbdi_state_cb_init()*.

5.16.3.3 Description

usbdi_pipe_state_get_req() is called by the LDD to get the state of the pipe described by ***pipe_channel.***

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

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

5.16.4 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.**

5.16.4.1 Synopsis

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

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

5.16.4.2 Arguments

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

5.16.4.3 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.

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

5.16.5 usbdi_pipe_state_get_ack_unused()

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

5.16.5.1 Synopsis

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

usbdi_pipe_state_get_ack_op_t usbdi_pipe_state_get_ack_unused;
```

5.16.5.2 Description

usbdi_pipe_state_get_ack_unused() maybe 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()*.

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

5.17 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()*

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

5.17.1 usbdi_edpt_state_set_req()

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

5.17.1.1 Synopsis

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

void usbdi_edpt_state_set_req(
            udi_channel_t pipe_channel,
            usbdi_state_cb_t *cb );
```

5.17.1.2 Arguments

udi_channel_t ***pipe_channel***	Channel for the pipe connecting the LDD to the endpoint.
usbdi_state_cb_t ****cb***	Pointer to a control block that was allocated with *udi_cb_alloc()* with the *cb_idx* value that was given to *usbdi_state_cb_init()*.

5.17.1.3 Description

usbdi_edpt_state_set_req() is called by the LDD to change the state of the USB endpoint described by ***pipe_channel***. 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)[11] for the endpoint associated with ***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.

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

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

5.17.2 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.**

5.17.2.1 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 );
```

5.17.2.2 Arguments

usbdi_state_cb_t ****cb***	Pointer to a control block containing data 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.

5.17.2.3 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.

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

5.17.3 usbdi_edpt_state_get_req()

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

5.17.3.1 Synopsis

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

void usbdi_edpt_state_get_req(
            udi_channel_t pipe_channel,
            usbdi_state_cb_t *cb);
```

5.17.3.2 Arguments

udi_channel_t ***pipe_channel***	Channel for the pipe connecting the LDD to the endpoint.
usbdi_state_cb_t ****cb***	Pointer to a control block that was allocated with *udi_cb_alloc()* with the *cb_idx* value that was given to *usbdi_state_cb_init()*.

5.17.3.3 Description

usbdi_edpt_state_get_req() is called by the LDD to get the state of the endpoint described by ***pipe_channel.*** In response to this call, the USBD issues a USB *GetStatus()*[12] 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.

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

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

5.17.4 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.**

5.17.4.1 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 );
```

5.17.4.2 Arguments

usbdi_state_cb_t ****cb***	Pointer to a control block containing data 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.

5.17.4.3 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.

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

5.17.5 usbdi_edpt_state_get_ack_unused()

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

5.17.5.1 Synopsis

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

usbdi_edpt_state_get_ack_op_t usbdi_edpt_state_get_ack_unused;
```

5.17.5.2 Description

usbdi_edpt_state_get_ack_unused() maybe 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().*

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

5.18 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()*

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

5.18.1 usbdi_intfc_state_set_req()

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

5.18.1.1 Synopsis

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

void usbdi_intfc_state_set_req(
            udi_channel_t intfc_channel,
            usbdi_state_cb_t *cb );
```

5.18.1.2 Arguments

udi_channel_t ***intfc_channel***	Channel for the interface.
usbdi_state_cb_t ****cb***	Pointer to a control block that was allocated with *udi_cb_alloc()* with the *cb_idx* value that was given to *usbdi_state_cb_init()*.

5.18.1.3 Description

usbdi_intfc_state_set_req() is called by the LDD to change the state of the interface described by ***intfc_channel***. 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.

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

5.18.2 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.**

5.18.2.1 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 );
```

5.18.2.2 Arguments

usbdi_state_cb_t ****cb***	Pointer to a control block containing data 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.

5.18.2.3 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.

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

5.18.3 usbdi_intfc_state_set_ack_unused()

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

5.18.3.1 Synopsis

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

usbdi_intfc_state_set_ack_op_t usbdi_intfc_state_set_ack_unused;
```

5.18.3.2 Description

usbdi_intfc_state_set_ack_unused() maybe 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()*.

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

5.18.4 usbdi_intfc_state_get_req()

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

5.18.4.1 Synopsis

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

void usbdi_intfc_state_get_req(
            udi_channel_t intfc_channel,
            usbdi_state_cb_t *cb);
```

5.18.4.2 Arguments

udi_channel_t ***intfc_channel***	Channel for the interface.
usbdi_state_cb_t ****cb***	Pointer to a control block that was allocated with *udi_cb_alloc()* with the *cb_idx* value that was given to *usbdi_state_cb_init()*.

5.18.4.3 Description

usbdi_intfc_state_get_req() is called by the LDD to get the state of the interface described by ***intfc_channel.*** The state of the interface will be returned to the LDD in *state* of CB. 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.

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

5.18.5 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.**

5.18.5.1 Synopsis

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

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

5.18.5.2 Arguments

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

5.18.5.3 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.

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

5.18.6 usbdi_intfc_state_get_ack_unused()

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

5.18.6.1 Synopsis

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

usbdi_intfc_state_get_ack_op_t usbdi_intfc_state_get_ack_unused;
```

5.18.6.2 Description

usbdi_intfc_state_get_ack_unused() maybe 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()*.

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

5.19 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()*

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

5.19.1 usbdi_device_state_get_req()

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

5.19.1.1 Synopsis

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

void usbdi_device_state_get_req(
            udi_channel_t bind_channel,
            usbdi_state_cb_t *cb);
```

5.19.1.2 Arguments

udi_channel_t ***bind_channel***	Bind channel for an interface controlled by the device whose state is being requested.
usbdi_state_cb_t ****cb***	Pointer to a control block that was allocated with *udi_cb_alloc()* with the *cb_idx* value that was given to *usbdi_state_cb_init()*.

5.19.1.3 Description

usbdi_device_state_get_req() is called by the LDD to get the state of the device described by ***bind_channel.*** The state of the device shall be returned to the LDD in *state* of the CB. 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.

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

5.19.2 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.**

5.19.2.1 Synopsis

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

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

5.19.2.2 Arguments

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

5.19.2.3 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.

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

5.19.3 usbdi_device_state_get_ack_unused()

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

5.19.3.1 Synopsis

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

usbdi_device_state_get_ack_op_t usbdi_device_state_get_ack_unused;
```

5.19.3.2 Description

usbdi_device_state_get_ack_unused() maybe 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().*

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

5.20 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_cb_init()*
- *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()*

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

5.20.1 usbdi_desc_cb_t

```
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;
```


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

5.20.2 usbdi_desc_cb_init()

***usbdi_desc_cb_init()* – Called at least once by the LDD from *init_module()* to set the requirements for *usbdi_desc_cb_t* if the LDD uses a *usbdi_desc_cb_t*.**

5.20.2.1 Synopsis

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

void usbdi_desc_cb_init(
           udi_index_t cb_idx,
           udi_size_t scratch_requirement );
```

5.20.2.2 Arguments

udi_index_t ***cb_idx***	Index value to be associated with the *usbdi_desc_cb_t*.
udi_size_t ***scratch_requirement***	Scratch space size (in bytes) needed by the LDD for the *usbdi_desc_cb_t*.

5.20.2.3 Description

The function *usbdi_desc_cb_init()* is called at least once by the LDD from *init_module()* if the LDD uses a *usbdi_desc_cb_t*. There is no callback or return value. The result is:

- The Management Agent associates the given ***cb_idx*** with the *usbdi_desc_cb_t*. This allows the LDD to allocate the *usbdi_desc_cb_t* by calling *udi_cb_alloc()* with an index value of ***cb_idx***.

- The scratch space size for the *usbdi_desc_cb_t* will be set to ***scratch_requirement.***

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

5.20.3 usbdi_desc_req()

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

5.20.3.1 Synopsis

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

void usbdi_desc_req(
            udi_channel_t intfc_channel,
            usbdi_desc_cb_t *cb );
```

5.20.3.2 Arguments

udi_channel_t **intfc_channel**	Channel for the interface.
usbdi_desc_cb_t ***cb**	Pointer to a control block that was allocated with *udi_cb_alloc()* with the *cb_idx* value that was given to *usbdi_desc_cb_init()*.

5.20.3.3 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()*[13] 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_BUF.

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.

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

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

5.20.4 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.**

5.20.4.1 Synopsis

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

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

5.20.4.2 Arguments

usbdi_desc_cb_t ****cb***	Pointer to a control block containing data 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.

5.20.4.3 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()*.

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

5.20.5 usbdi_desc_ack_unused()

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

5.20.5.1 Synopsis

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

usbdi_desc_ack_op_t usbdi_desc_ack_unused;
```

5.20.5.2 Description

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

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

5.21 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()*

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

5.21.1 usbdi_config_set_req()

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

5.21.1.1 Synopsis

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

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

5.21.1.2 Arguments

udi_channel_t ***intfc_channel***	Channel of the interface.
usbdi_misc_cb_t ****cb***	Pointer to a control block allocated with *udi_cb_alloc()* with the *cb_idx* value that was given to *usbdi_misc_cb_init()*.
udi_ubit16_t **config_value**	New configuration value.

5.21.1.3 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()*[14] 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.

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

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

5.21.2 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.**

5.21.2.1 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 );
```

5.21.2.2 Arguments

usbdi_desc_cb_t ****cb***	Pointer to a control block containing data 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.

5.21.2.3 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.

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

5.21.3 usbdi_config_set_ack_unused()

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

5.21.3.1 Synopsis

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

usbdi_config_set_ack_op_t usbdi_config_set_ack_unused;
```

5.21.3.2 Description

usbdi_config_set_ack_unused() maybe 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()*.

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

5.22 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 5.17.5, "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*

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

5.22.1 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.**

5.22.1.1 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);
```

5.22.1.2 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.

5.22.1.3 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()*.

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

5.22.2 usbdi_async_event_ind_unused()

usbdi_async_event_ind_unused() **- Used by the LDD to specify to the USBD that it shall never receive a** ***usbdi_async_event_req.***

5.22.2.1 Synopsis

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

usbdi_async_event_ind_op_t usbdi_async_event_ind_unused;
```

5.22.2.2 Description

usbdi_async_event_ind_unused() maybe 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.*

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

5.22.3 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.**

5.22.3.1 Synopsis

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

void usbdi_async_event_res(
            udi_channel_t intfc_channel,
            usbdi_misc_cb_t *cb );
```

5.22.3.2 Arguments

udi_channel_t ***intfc_channel***	Channel of the interface
usbdi_misc_cb_t ****cb***	Pointer to a control block containing data passed to the LDD's *usbdi_async_event_ind_op_t* function.

5.22.3.3 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.

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

6 UDI Overview

This section briefly describes some of the UDI abstractions, functions, and control blocks that OpenUSBDI depends on. This section is for reference only; the UDI Core Specification is the final authority in the event of a conflict. The following UDI topics are covered in this overview:

- Management Agent
- *udi_cb_alloc()*
- *udi_cb_free()*
- *udi_channel_spawn()*
- *udi_channel_close()*
- *udi_cb_t*

6.1 Management Agent

The Management Agent (MA) is an abstract entity within the UDI environment; it represents the environment's control and configuration mechanisms. All device configuration, driver instantiation, and initial channel creation are driven and controlled by the MA. The MA is an integral part of the environment and is always present in any UDI environment implementation.

The Management Metalanguage defines the communication between a driver instance and the MA and is used for the *management channel*, which is a channel that is always present between the MA and the driver's primary region[15]. When the MA wishes to instruct a driver instance to perform a management-related operation it will generate a Management Metalanguage request to that driver instance over its management channel. The driver will respond via a Management Metalanguage acknowledgement.

Additionally, the driver may indicate system-level events by generating responses to corresponding Management Metalanguage inquiries, but the driver shall never initiate an operation to the MA.

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

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

6.2 udi_cb_alloc()

Allocate a new control block.

6.2.1 Synopsis

```
#include <udi.h>

void udi_cb_alloc(
            udi_cb_alloc_call_t *callback,
            udi_cb_t *gcb,
            udi_index_t cb_idx );

typedef void udi_cb_alloc_call_t(
            udi_cb_t *gcb,
            udi_cb_t *new_cb );
```

6.2.2 Arguments

udi_cb_alloc_call_t ***callback**	LDD function called once the CB has been allocated.
udi_cb_t ***gcb**	Control block that the LDD received that initiated the need for a new CB.
udi_index_t ***cb_idx***	Control block index that indicates required properties of the control block.
udi_cb_t ***new_cb**	Pointer to the newly allocated control block.

6.2.3 Description

Allocate a new control block for use by the driver. The new control block may be used to allocate other resources using any UDI service request or to invoke channel operations appropriate to the specified control block type.

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

6.3 udi_cb_free()

Deallocates a previously obtained control block.

6.3.1 Synopsis

```
#include <udi.h>

void udi_cb_free(
            udi_cb_t *cb )
```

6.3.2 Arguments

udi_cb_t ***cb***	Pointer to control block to be deallocated. If NULL, this function is a no-op.

6.3.3 Description

Releases the specified control block, including any metalanguage-specific parts, along with any associated resources back to the environment; CB shall be NULL or shall have been previously obtained by a call to *udi_cb_alloc()*, or passed to the driver via a channel operation.

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

6.4 udi_channel_spawn()

Spawn a new channel with loose ends.

6.4.1 Synopsis

```
#include <udi.h>

void udi_channel_spawn(
            udi_channel_spawn_call_t *callback,
            udi_cb_t *gcb,
            udi_channel_t channel,
            udi_index_t spawn_idx,
            udi_index_t ops_idx,
            void *channel_context);

typedef void udi_channel_spawn_call_t (
            udi_cb_t *gcb,
            udi_channel_t new_channel );
```

6.4.2 Arguments

udi_cb_alloc_call_t ***callback**	LDD function called once the CB has been allocated.
udi_cb_t ****gcb***	Control block that the LDD received that initiated the need for a new channel.
udi_channel_t ***channel***	Channel handle for an existing anchored channel. The channel will be spawned relative to this channel.
udi_index_t ***spawn_idx***	Small integer which allows the environment to match two spawn requests (one from each end of the channel) together.
udi_index_t ***ops_idx***	Ops index for the ops vector that the driver wants to associate with the specified channel, passed to the appropriate "*ops_init()*" at *init_module()* time, or zero.
void ****channel_context***	Channel context pointer to be associated with the new anchored channel endpoint.
udi_channel_t ****new_channel***	Channel handle for the new channel's local endpoint, which will be a loose end. This handle shall subsequently be passed to *udi_channel_anchor()*, either in this region, or after passing it to another region via a channel operation.

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

6.4.3 Description

Used to create a new channel (initially) between the same two regions as an existing channel. Both ends shall be created separately by their own calls to *udi_channel_spawn().*

The pair of the original channel handle and the spawn index uniquely identifies an in-progress spawn operation. Once both regions have rendezvoused inside a spawn request, the new channel is created and each driver is passed its handle for the new channel via the ***callback***.

If ***ops_idx*** is zero, the channel endpoint is created as a loose end, which shall be anchored before it may be used. Loose ends may be passed between regions, and even between drivers, before being anchored.

It is the LDD's responsibility to ensure that both ends are anchored and ready to go before invoking any operations on the channel. This is typically done via metalanguage-specific handshaking on the original channel.

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

6.5 udi_channel_close()

Close a channel.

6.5.1 Synopsis

```
#include <udi.h>

void udi_channel_close(
            udi_channel_t channel );
```

6.5.2 Arguments

udi_channel_t ****channel***	Channel handle for the channel being closed.

6.5.3 Description

Deallocates and returns any channel related resources to the UDI environment. Normally a driver calls this routine only as a result of receiving a *udi_channel_event_ind()* operation of type UDI_CHANNEL_CLOSED, to close its end of the channel.

The result of this routine is immediate: the channel endpoint will be closed and freed when this call returns. It is the responsibility of the driver to clean up all channel related state and resources first, so as to maintain architectural integrity before destroying a channel. This shall include the processing of all outstanding operations related to the channel. The driver may ensure, via proper channel operation handling, that all operations directed to this channel have been completed and that no more will be generated. Any operations previously sent to this channel but not yet delivered at the time this routine is called will be treated as having been initiated after the channel was closed.

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

6.6 udi_cb_t

Generic, least-common-denominator control block.

6.6.1 Synopsis

```
#include <udi.h>

typedef struct {
            void *context;
            void *scratch;
} udi_cb_t;
```

6.6.2 Members

void ***context**	Pointer to state information within the driver region. On entry to a channel operation, the environment sets ***context*** to the channel's current context. Drivers may change it if needed.
void ***scratch**	Pointer to the control block's scratch area. Drivers shall not change this pointer, but may change any of the bytes in the space pointed to by ***scratch***, up to the scratch size specified by each driver specified by the specific "*cb_init()*" function..

6.6.3 Description

The *udi_cb_t* structure is used for generically handling control blocks and accessing their scratch area and context pointer. All metalanguage-specific control blocks have a *udi_cb_t* structure as their first member.

Both ***`context`*** and ***`scratch`***, as well as the scratch area contents, are preserved across asynchronous service calls, but not across interface operations.

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

7 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.

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

8 Outstanding Issues

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

9 Appendix A: Include File (open_usbdi.h)

```
#ifndef _OPEN_USBDI_H_
#define _OPEN_USBDI_H_

#if OPEN_USBDI_VERSION = 0x080

/*
 * 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
                               */
```


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

```
#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 */
```


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

```
    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
```


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

```
                              * 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
```


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

```
    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.
 *
```


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

```
 * 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 CB (Control Block) there is an associated set of functional
 * interface declarations.  These functions define how the LDD (logical
 * device driver) and the USBD (USB protocol driver) communicate.  The
 * following five functions are provided for each CB:
 *
 * usbdi_XX_cb_init
 *     Called by the LDD only once, when the LDD is initialized.  It
 *     sets up LDD specific values for the CB such as the required
 *     scratch size.
 *
 * 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).
```


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

```
 */

/*
 * 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;

void usbdi_misc_cb_init(udi_index_t cb_idx,
                       udi_size_t scratch_requirement);

/*
 * 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(       udi_channel_t bind_channel,
                                     usbdi_misc_cb_t *cb);

typedef void   usbdi_bind_req_op_t(   usbdi_misc_cb_t *cb);

void           usbdi_bind_ack(       udi_channel_t channel,
                                     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(     udi_channel_t bind_channel,
                                     usbdi_misc_cb_t *cb);

typedef void   usbdi_unbind_req_op_t( usbdi_misc_cb_t *cb);

void           usbdi_unbind_ack(     udi_channel_t channel,
                                     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.
 */
```


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

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

typedef void usbdi_intfc_open_req_op_t(usbdi_misc_cb_t *cb);

void             usbdi_intfc_open_ack(udi_channel_t intfc_channel,
                                     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(udi_channel_t intfc_channel,
                                      usbdi_misc_cb_t *cb);

typedef void usbdi_intfc_close_req_op_t(usbdi_misc_cb_t *cb);

void             usbdi_intfc_close_ack(udi_channel_t intfc_channel,
                                      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 */
    void *tr_context;  /* Transaction specific context info */

    /*
     * data_buf and data_len
     *
     * data_buf points to the associated buffer where the data
     * will be transferred to/from.  The LDD sets the data_len field
     * to the maximum number of bytes which may be transferred.  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;
    udi_size_t data_len;

    /*
     * 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;
```


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

```
    /*
     * 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)

} usbdi_intr_bulk_xfer_cb_t;

void           usbdi_intr_bulk_xfer_cb_init( udi_index_t cb_idx,
                                             udi_size_t scratch_requirement);

void           usbdi_intr_bulk_xfer_req(     udi_channel_t pipe_channel,
                                             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(     udi_channel_t pipe_channel,
                                             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(     udi_channel_t pipe_channel,
                                             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 */
    void *tr_context;  /* Transaction specific context info */

    /*
     * 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
```


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

```
     * 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;
    udi_size_t data_len;

    /*
     * 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;
#define USBDI_XFER_IN  (1<<2) /* transfer data from the device */
#define USBDI_XFER_OUT (1<<3) /* transfer data to the device */

} usbdi_control_xfer_cb_t;

void           usbdi_control_xfer_cb_init(   udi_index_t cb_idx,
                                              udi_size_t scratch_requirement);

void           usbdi_control_xfer_req(       udi_channel_t pipe_channel,
                                              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(       udi_channel_t pipe_channel,
                                              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(       udi_channel_t pipe_channel,
                                              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_size_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.
                               */
```


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

```
    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 */
    void *tr_context;  /* Transaction specific context info */

    /*
     * 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<<4)

    /*
     * 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;

void           usbdi_isoc_xfer_cb_init(      udi_index_t cb_idx,
                                             udi_size_t scratch_requirement,
                                             udi_ubit8_t frame_array_size);

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);
```


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

```
void            usbdi_isoc_xfer_ack(          udi_channel_t pipe_channel,
                                              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(          udi_channel_t pipe_channel,
                                              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(       udi_channel_t intfc_channel,
                                              usbdi_misc_cb_t *cb);

typedef void    usbdi_frame_number_req_op_t(  usbdi_misc_cb_t *cb);

void            usbdi_frame_number_ack(       udi_channel_t intfc_channel,
                                              usbdi_misc_cb_t *cb,
                                              udi_ubit32_t frame_number,
                                              udi_status_t status);

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

usbdi_frame_number_ack_op_t                   usbdi_frame_number_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(       udi_channel_t intfc_channel,
                                              usbdi_misc_cb_t *cb);

typedef void    usbdi_reset_device_req_op_t(  usbdi_misc_cb_t *cb);

void            usbdi_reset_device_ack(       udi_channel_t intfc_channel,
                                              usbdi_misc_cb_t *cb,
                                              udi_status_t status);

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

usbdi_reset_device_ack_op_t                   usbdi_reset_device_ack_unused;

/*
 * Abort a Transfer.
 *
 * The aborted transfer request will be returned via its callback
 * function.  Only after the transfer request has been aborted will
```


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

```
 * "usbdi_xfer_abort_ack_op" be called.  The associated pipe channel's
 * state will be USBDI_STATE_STALLED on completion of the abort process.
 */
typedef struct {
       udi_cb_t gcb;
       void *tr_context;
       void *orig_tr_context;
} usbdi_xfer_abort_cb_t;

void           usbdi_xfer_abort_cb_init(     udi_index_t cb_idx,
                                             udi_size_t scratch_requirement);

void           usbdi_xfer_abort_req(         udi_channel_t pipe_channel,
                                             usbdi_xfer_abort_cb_t *cb);

typedef void   usbdi_xfer_abort_req_op_t(    usbdi_xfer_abort_cb_t *cb);

void           usbdi_xfer_abort_ack(         udi_channel_t pipe_channel,
                                             usbdi_xfer_abort_cb_t *cb,
                                             udi_status_t status);

typedef void   usbdi_xfer_abort_ack_op_t(    usbdi_xfer_abort_cb_t *cb,
                                             udi_status_t status);

usbdi_xfer_abort_ack_op_t usbdi_xfer_abort_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 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(        udi_channel_t intfc_channel,
                                             usbdi_misc_cb_t *cb);
```


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

```
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;

void   usbdi_state_cb_init(   udi_index_t cb_idx,
                              udi_size_t scratch_requirement);

/*
 * 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(     udi_channel_t pipe_channel,
                                             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(     udi_channel_t pipe_channel,
                                             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(     udi_channel_t pipe_channel,
                                             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(     udi_channel_t pipe_channel,
                                             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
 *
```


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

```
 *    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(     udi_channel_t pipe_channel,
                                              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(     udi_channel_t pipe_channel,
                                              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(     udi_channel_t pipe_channel,
                                              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(     udi_channel_t pipe_channel,
                                              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(    udi_channel_t intfc_channel,
                                              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(    udi_channel_t intfc_channel,
                                              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(    udi_channel_t intfc_channel,
                                              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(    udi_channel_t intfc_channel,
```


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

```
                                             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(          udi_channel_t bind_channel,
                                                    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(          udi_channel_t bind_channel,
                                                    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;
```


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

```
#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;

void           usbdi_desc_cb_init(    udi_index_t cb_idx,
                                      udi_size_t scratch_requirement);

void           usbdi_desc_req(                udi_channel_t intfc_channel,
                                      usbdi_desc_cb_t *cb);

typedef void   usbdi_desc_req_op_t(   usbdi_desc_cb_t *cb);

void           usbdi_desc_ack(                udi_channel_t intfc_channel,
                                      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(                 udi_channel_t intfc_channel,
                                              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.
 *
```


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

```
 * 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(        udi_channel_t intfc_channel,
                                             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_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;

void   usbdi_ldd_intfc_ops_init(     udi_index_t ops_index,
                                     usbdi_ldd_intfc_ops_t *ops);

/*
 * 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_xfer_abort_ack_op_t            *usbdi_xfer_abort_ack_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;
```


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

```
        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;

void    usbdi_ldd_pipe_ops_init(              udi_index_t ops_index,
                                              usbdi_ldd_pipe_ops_t *ops);

/*
 * 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;

void    usbdi_usbd_intfc_ops_init(            udi_index_t ops_index,
                                              usbdi_usbd_intfc_ops_t *ops);

/*
 * 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_xfer_abort_req_op_t            *usbdi_xfer_abort_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;

void    usbdi_usbd_pipe_ops_init(             udi_index_t ops_index,
                                              usbdi_usbd_pipe_ops_t *ops);
#endif /* OPEN_USBDI_VERSION */
#endif /* _OPEN_USBDI_H_ */
```


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

10 Appendix B: Sample Driver (usbdi_printer.c)

```
#define OPEN_USBDI_VERSION 0x08
#include <udi.h>

#include <open_usbdi.h>

#include "usbdi_printer.h"



/*
 * 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 .8)
 * and UDI (ver .90) 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.
 */

static udi_mgmt_ops_t print_mgmt_ops = {
    udi_region_attach_unused,
    print_gio_prep_for_child_req,
    print_usbdi_bind_to_parent_req,
    print_usbdi_unbind_from_parent_req,
    print_enumerate_req,
    print_trace_mod_req
};

static 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,
    print_gio_abort_req,
    print_gio_event_res
};

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_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
};
```


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

```
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,
    print_usbdi_xfer_abort_ack,
    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
};

/*
 * All USBDI (and all UDI) driver modules shall contain a main
 * initialization driver entry point, init_module().
 */
void
init_module(void)
{
    /*
     * Set up the scratch sizes for the various USBDI control blocks
     * that this driver will use.
     */
    usbdi_misc_cb_init(          PRINT_USBDI_MISC_CB_IDX,
                                PRINT_USBDI_MISC_CB_SCRATCH_SIZE);
    usbdi_misc_cb_init(          PRINT_USBDI_OPEN_CLOSE_CB_IDX,
                                PRINT_USBDI_OPEN_CLOSE_CB_SCRATCH_SIZE);
    usbdi_intr_bulk_xfer_cb_init(PRINT_USBDI_BULK_XFER_CB_IDX,
                                PRINT_USBDI_BULK_XFER_CB_SCRATCH_SIZE);
    usbdi_control_xfer_cb_init(  PRINT_USBDI_CONTROL_XFER_CB_IDX,
                                PRINT_USBDI_CONTROL_XFER_CB_SCRATCH_SIZE);
    usbdi_xfer_abort_cb_init(    PRINT_USBDI_XFER_ABORT_CB_IDX,
                                PRINT_USBDI_XFER_ABORT_CB_SCRATCH_SIZE);
    usbdi_state_cb_init(         PRINT_USBDI_STATE_CB_IDX,
                                PRINT_USBDI_STATE_CB_SCRATCH_SIZE);
    usbdi_desc_cb_init(          PRINT_USBDI_DESC_CB_IDX,
                                PRINT_USBDI_DESC_CB_SCRATCH_SIZE);

    /*
     * Register the driver's OpenUSBDI interface related Ops.
     */
    usbdi_ldd_intfc_ops_init(PRINT_USBDI_INTFC_OPS_IDX,
                            &print_usbdi_ldd_intfc_ops);

    /*
     * Register the driver's OpenUSBDI pipe related Ops.
     */
    usbdi_ldd_pipe_ops_init(PRINT_USBDI_PIPE_OPS_IDX,
                           &print_usbdi_ldd_pipe_ops);

    /*
     * Set up the LDDs scratch sizes for the GIO control blocks.
     */
    udi_gio_bind_cb_init(       PRINT_GIO_BIND_CB_IDX,
                                PRINT_GIO_BIND_CB_SCRATCH_SIZE);
    udi_gio_unbind_cb_init(     PRINT_GIO_UNBIND_CB_IDX,
                                PRINT_GIO_UNBIND_CB_SCRATCH_SIZE);
    udi_gio_xfer_cb_init(       PRINT_GIO_XFER_CB_IDX,
                                PRINT_GIO_XFER_CB_SCRATCH_SIZE,
                                PRINT_GIO_XFER_CB_PARAMS_SIZE);
    udi_gio_abort_cb_init(      PRINT_GIO_ABORT_CB_IDX,
                                PRINT_GIO_ABORT_CB_SCRATCH_SIZE);
    udi_gio_event_cb_init(      PRINT_GIO_EVENT_CB_IDX,
```


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

```
                                PRINT_GIO_EVENT_CB_SCRATCH_SIZE);

    /*
     * This LDD is a GIO provider.  Register the GIO related ops.
     */
    udi_gio_provider_interface_init(PRINT_GIO_OPS_IDX,
                                    &print_gio_provider_ops);

    /*
     * Initialize this driver's primary region.
     */
    udi_primary_region_init(&print_mgmt_ops,
                            NULL,
                            PRINT_GIO_OPS_IDX,
                            PRINT_USBDI_INTFC_OPS_IDX,
                            0,
                            sizeof(struct printer_context),
                            NULL);
}

void
print_gio_prep_for_child_req(udi_bind_cb_t *cb,
                             void *enumeration_context)
{
    /*
     * The printer_context struct was allocated by UDI.  The
     * size had been given to udi_primary_region_init() in
     * init_module().
     */
    struct printer_context *printer = cb->gcb.context;

    /*
     * Store away the bind_channel.  This will be the channel
     * used for all GIO operations.
     */
    printer->gio_bind_channel = cb->bind_channel;

    /*
     * Associate the enumeration_context with this printer_context struct.
     */
    printer->enumeration_context = enumeration_context;

    /*
     * Since child_bind_interface passed to udi_primary_region_init()
     * above was not zero, the bind_channel has already been anchored.
     */

    /*
     * Acknowledge this request.
     */
    udi_prep_for_child_ack(cb->bind_channel, cb, UDI_OK);
}

/*
 * Function:  print_usbdi_bind_to_parent_req()
 *
 * This function is called by the UDI Management Agent when the
 * usbdi_printer driver (the printer LDD) has been requested to
 * bind to an interface group by the USBD.  In this case the LDD
 * will only be bound to a single interface (interface group size
 * is one interface).
 */
static void
print_usbdi_bind_to_parent_req(udi_bind_cb_t *cb)
{
    /*
     * The printer_context struct was allocated by UDI.  The
     * size had been given to udi_primary_region_init() in
```


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

```
     * init_module().
     */
    struct printer_context *printer = cb->gcb.context;

    /*
     * Store away the bind_channel.  This will be the channel
     * used for the interface that the LDD is being bound to.
     * This is also the channel used for all requests issued
     * to the default endpoint of the printer.
     */
    printer->usbdi_intfc_channel = cb->bind_channel;
    printer->udi_bind_cb = cb;

    /*
     * Allocate a usbdi_misc_cb_t for usbdi_bind_req.  This struct will
     * be used to complete the binding.  Execution continues in
     * print_usbdi_bind_cb_alloc_call().
     */
    udi_cb_alloc(print_usbdi_bind_cb_alloc_call, UDI_GCB(cb),
                PRINT_USBDI_MISC_CB_IDX);
}

/*
 * Function: print_bind_cb_alloc_call()
 *
 * Callback function given to udi_cb_alloc() in
 * print_usbdi_bind_to_parent_req().
 */
static void
print_usbdi_bind_cb_alloc_call(udi_cb_t *gcb, udi_cb_t *new_cb)
{
    udi_bind_cb_t *udi_bind_cb = UDI_MCB(gcb, udi_bind_cb_t);
    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(udi_bind_cb->bind_channel, usbdi_bind_cb);

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

/*
 * 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->udi_bind_cb),
```


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

```
                PRINT_USBDI_DESC_CB_IDX);

    /*
     * 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(printer->usbdi_intfc_channel, 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.
     */
}

/*
 * 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;
```


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

```
        cb->desc_ID = 0;
        cb->desc_length = sizeof(struct usb_interface_descriptor);
        usbdi_desc_req(printer->usbdi_intfc_channel, cb);
        return;

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

    /*
     * 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(printer->usbdi_intfc_channel, 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_bind_to_parent_ack(printer->usbdi_intfc_channel,
                           printer->udi_bind_cb, NULL, UDI_OK);
    printer->udi_bind_cb = (udi_bind_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().
     */
}
```


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

```
/* JANET - Still need to do unbind */
void
print_usbdi_unbind_from_parent_req(udi_unbind_cb_t *cb,
                                   void *bind_context,
                                   udi_ubit8_t flags)
{
}

/* JANET - Still need to do enumerate */
void
print_enumerate_req(udi_enumerate_cb_t *cb,
                    udi_ubit8_t enumeration_level)
{
}

void
print_trace_mod_req(udi_trace_cb_t *cb)
{
}

static void
print_gio_provider_channel_event_ind(udi_channel_event_ind_cb_t *cb)
{
}

/*
 * 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;
    printer->gio_bind_cb = gio_bind_cb;

    /*
     * 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(printer->gio_bind_channel, gio_bind_cb,
                        0, 0, UDI_STAT_BUSY);
       return;
    }

    /*
     * Execution continues in print_usbdi_intfc_open.
     */
    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 to 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;
```


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

```
    /*
     * Make sure that a bind has already been done.
     */
    if (!(printer->flags & PRINT_FLAGS_GIO_BOUND)) {
        udi_gio_bind_ack(printer->gio_bind_channel, gio_bind_cb,
                         0, 0, UDI_STAT_CANNOT_UNBIND);
        return;
    }

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

    /*
     * Make sure that outstanding requests complete before the
     * interface is closed.
     * JANET - todo
     */

    /*
     * We make it here if there are no outstanding requests
     * and we are ready to close the interface.
     */
    printer->flags |= PRINT_FLAGS_GIO_UNBOUND;
    print_usbdi_intfc_close(UDI_GCB(gio_bind_cb));
}

/*
 * 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 uid_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 = gio_cb->gcb.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->data_dir != UDI_GIO_DIR_WRITE) {
        udi_gio_xfer_ack(printer->gio_bind_channel, 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_ack(printer->gio_bind_channel, 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;
```


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

```
    /*
     * Make the tr_context point to the gio_cb.  The gio_cb will be the
     * cookie that we will use to abort the request if needed.
     */
    bulk_cb->tr_context = (void *)gio_cb;
    bulk_cb->data_buf = gio_cb->data_buf;
    bulk_cb->data_len = gio_cb->data_len;

    /*
     * Send it on down.
     */
    usbdi_intr_bulk_xfer(pipe->channel, bulk_cb);

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

static void print_gio_abort_req(udi_gio_abort_cb_t *cb)
{
}

void print_gio_event_res(udi_gio_event_cb_t *cb)
{
}

static void
print_usbdi_ldd_intfc_channel_event_ind(udi_channel_event_ind_cb_t *cb)
{
}

/*
 * 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.
     */
    printer->usbdi_bind_cb = cb;

    /*
     * 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_bind_to_parent_ack(printer->usbdi_intfc_channel,
                              printer->udi_bind_cb, NULL,
                              UDI_STAT_CANNOT_BIND);
        printer->udi_bind_cb = (udi_bind_cb_t *)NULL;
        print_free_printer(printer);
        return;
    }
```


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

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

/*
 * 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_UNBOUND;

    udi_gio_unbind_ack(printer->gio_bind_channel, printer->gio_bind_cb);
    printer->gio_bind_cb = (udi_gio_bind_cb_t *)NULL;
}

static void
print_usbdi_intfc_abort_ack(usbdi_misc_cb_t *cb)
{
}
```


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

```
/*
 * 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_bind_to_parent_ack(printer->usbdi_intfc_channel,
                              printer->udi_bind_cb, NULL,
                              UDI_STAT_CANNOT_BIND);
        printer->udi_bind_cb = (udi_bind_cb_t *)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;

    /*
     * 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);
}

static void
print_usbdi_async_event_ind(usbdi_misc_cb_t *cb,
                           udi_ubit16_t async_event)
{
}

static void
print_usbdi_ldd_pipe_channel_event_ind(udi_channel_event_ind_cb_t *cb)
{
}
```


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

```
/*
 * 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 = bulk_cb->tr_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;

    /*
     * Update the data_len of the udi_gio_xfer_cb.
     */
    gio_cb->data_len = bulk_cb->data_len;
    udi_gio_xfer_ack(printer->gio_bind_channel, gio_cb, UDI_OK);
    return;
}

/*
 * Function: print_usbdi_intr_bulk_xfer_nak()
 *
 * Called in 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 = bulk_cb->tr_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;

    /*
     * 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.
```


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

```
         */
        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:
    default:
        /*
         * Need to add support for these error conditions.  For now,
         * fall through to UDI_STAT_INVALID_STATE.
         */

    case UDI_STAT_INVALID_STATE:
        /*
         * 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_len = 0;
        udi_gio_xfer_ack(printer->gio_bind_channel,
                         printer->gio_xfer_cb, status);
    }

    /*
     * In all 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)
{
}

static void
print_usbdi_xfer_abort_ack(usbdi_xfer_abort_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
```


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

```
 * 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);
        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_channel,
                        printer->usbdi_intfc_open_close_cb,
                        printer->usb_interface_descriptor->bAlternateSetting);

    /*
     * 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)
{
```


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

```
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.
```


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

```
     */
    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;
```


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

```
    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.
         */
        udi_cb_alloc(print_usbdi_intr_bulk_cb_alloc_call,
                     UDI_GCB(printer->udi_bind_cb),
                     PRINT_USBDI_BULK_XFER_CB_IDX);
        return;

        /*
         * Execution continues with print_usbdi_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_channel,
                     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.
 */
```


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

```
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_channel,
                          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.
     */
```


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

```
    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);
        return;

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

    cb->state = USBDI_STATE_ACTIVE;
    usbdi_pipe_state_set_req(pipe->channel, 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.
     */
```


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

```
    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);
}

static void
print_usbdi_edpt_state_cb_alloc_call(udi_cb_t *gcb, udi_cb_t *cb)
{
```


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

```
    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(pipe->channel, 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));
}

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)
```


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

```
{
}
```


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

11 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;

    udi_channel_t gio_bind_channel; /* Channel use to communicate with the
                                     * GIO child
                                     */
    udi_channel_t usbdi_bind_channel; /* Channel used to communicate with
                                     * the USBD when performing device
                                     * related ops
                                     */
    udi_channel_t usbdi_intfc_channel;/* Channel used to communicate with the
                                     * USBD when performing intfc related
                                     * ops
                                     */
    udi_bind_cb_t *udi_bind_cb;       /* CB given to
                                     * print_bind_to_parent_req()
                                     */
    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.
                                     */
```


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

```
    /*
     * 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_XFER         (1<<3)
#define PRINT_FLAGS_PIPE_STATE               (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;
```


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

```
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
 */
static udi_prep_for_child_req_op_t print_gio_prep_for_child_req;
static udi_bind_to_parent_req_op_t print_usbdi_bind_to_parent_req;
static udi_unbind_from_parent_req_op_t print_usbdi_unbind_from_parent_req;
static udi_enumerate_req_op_t print_enumerate_req;
static udi_trace_mod_req_op_t print_trace_mod_req;

/*
 * 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_xfer_abort_ack_op_t print_usbdi_xfer_abort_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;
static udi_gio_abort_req_op_t print_gio_abort_req;
```


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

```
static udi_gio_event_res_op_t print_gio_event_res;

/*
 * 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_XFER_ABORT_CB_IDX       4
#define PRINT_USBDI_STATE_CB_IDX                    5
#define PRINT_USBDI_DESC_CB_IDX             6
#define PRINT_USBDI_OPEN_CLOSE_CB_IDX       7

/*
 * 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_XFER_ABORT_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_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_ABORT_CB_IDX                      4
#define PRINT_GIO_EVENT_CB_IDX                      5

/*
 * 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_ABORT_CB_SCRATCH_SIZE     sizeof(void *)
#define PRINT_GIO_EVENT_CB_SCRATCH_SIZE     sizeof(void *)
```


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