MCCI Corporation
3520 Krums Corners Road
Ithaca, New York 14850 USA
Phone +1-607-277-1029
Fax +1-607-277-6844
www.mcci.com

MCCI USB DataPump Embedded USBDI

Engineering Report 950000325
Rev. E
Date: 2011/09/28

PROPRIETARY NOTICE AND DISCLAIMER

Document Release History

Rev. A	2007/07/08	Original release
Rev. B	2007/12/21	Additions
Rev. C	2008/01/08	Added new PORT IOCTLs
Rev. D	2011/03/28	Changed document numbers to nine digit versions. DataPump 3.0 Updates.
Rev. E	2011/09/28	Added source code disclaimer.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 Introduction

1.1 Purpose

This document describes the USB Driver Interface (USBDI) API provided by the MCCI USB DataPump Embedded Host / On-The-Go USB host stack.

1.2 Scope

USBDI is the industry-standard reference term for the API between USB device drivers and the lower levels of the USB host stack. This document describes the API as used by class drivers. Descriptions of the implementation of the USBD stack are informative, not normative.

This document assumes familiarity with the MCCI USB DataPump.

1.3 Glossary

Brand	MCCI's term for the concrete set of drivers derived from the MCCI core library with changes as specified by the customer
Composite device	A specific way of representing a USB device that supports multiple independent functions concurrently. In this model, each USB Function consists of one or more interfaces, with the associated endpoints and descriptors. On Windows, the parent driver divides the composite device up into single functions, and then uses standard object-oriented techniques to present the descriptors of each function to the function drivers. This allows function drivers to be coded the same way whether they are running as the sole function on a device or as part of a multi-function composite device. Compare with "compound device" as defined in [USBCORE].
DCD	*See* Device Controller Driver
Device controller	The hardware module responsible for connecting a USB device to the USB bus.
Device Controller Driver (DCD)	The software component that provides low-level access to the specific Device Controller in use. All MCCI USB DataPump DCDs implement a common API, allowing the rest of the DataPump device stack to be hardware independent.
Device stack	Collective term for the software stack that implements USB device functionality.
EH	Embedded Host

HCD	*See* Host Controller Driver
HCD Class	*See* Host Controller Driver
HCD Instance	*See* Host Controller Driver
Host controller	The hardware module responsible for operating the USB bus as a host.
Host Controller Driver	The software component that provides low–level access to the specific Host Controller in use. This term may refer a specific instance of the software that models the host controller to upper layers of software, or it may refer to the entire collection of code that implements the driver. Where necessary, we refer to the collection of code as the "HCD Class", and the specific data structures and methods that represent a given instance as an "HCD Instance".
OTG	Abbreviation for USB On-The-Go
OTGCD	*See* OTG Controller Driver
OTG Controller	The hardware module responsible for operating a dual-role OTG connection.
OTG Controller Driver	The software component that provides low-level access to a USB bus via an OTG Controller. Normally export three APIs, an HCD API, a DCD API, and a (shared) OTG
Phy	Short for "physical layer". Often used as short-hand for "transceiver". MCCI uses this in the abbreviations for the API operatons that are used for accessing the phy.
Transceiver	The hardware module responsible for low-level signaling on the USB bus.
USBD	USB Driver, the generic term for the USB Management module.
USBDI	USB Driver Interface, the generic term for the API between USB function drivers and USBD.
xCD	Host, Device, Dual-Role or OTG Controller Driver

1.4 Referenced Documents

[EHUG] *MCCI USB DataPump Embedded Host and OTG Users Guid*e, MCCI Engineering report 950000327

[Knuth1973] *The Art of Computer Programming* **1** (Second Edition), Reading MA, Addison Wesley

[USBCORE] *Universal Serial Bus Specification*, version 2.0/3.0 (also referred to as the USB

Specification), with published erratas and ECOs. This specification is available on the World Wide Web site http://www.usb.org/.

[DPUSERGUIDE] *MCCI USB Datapump User's Guide,* MCCI Engineering Report 950000066

[USBRC] *USBRC User's Guide,* MCCI Engineering Report 950000061

2 Overview

The MCCI EH USBDI interfaces provide a portable, efficient means of writing host device drivers for USB devices using the MCCI Embedded Host stack.

The DataPump USBDI operates as part of the MCCI USB DataPump. As such, the programming environment assumes a fairly simple model for tasking and reentrancy control. MCCI USBDI drivers are not tasks – instead they are asynchronous finite state machines, similar to the class protocols implemented by the USB device DataPump.

- Class Drivers are not allowed to call blocking APIs. If they block, then the USB subsystem will stop making progress.

- Instead, APIs that would block are asynchronous. The class driver prepares a request block of some kind, containing a pointer to a call back function. Calls that submit the request to another module return without waiting for the operation to complete. Later, when the operation completes, the call-back function is invoked.

- Despite the asynchronous APIs, the possible concurrency is limited. The DataPump processes external events (whether from other tasks or from hardware interrupts) by appending notifications to the DataPump event queue. This queue is processed sequentially. Therefore, a callback routine can never preempt other DataPump activity, nor can it be preempted. We justify the lack of concurrency by noting that USB is a serial bus, and typically one has only one CPU to manage all the USB ports. Furthermore, the serialization enforces a degree of fairness for processing of incoming requests.

Device drivers are designed with a "class/instance" model. During DataPump initialization, the class initialization entry point is called. This initialization entry point creates a "class object" which models the driver, and one or more "instance objects", which model specific instances of the device. Class objects are derived from the `USBPUMP_USBDI_DRIVER_CLASS` type; and instance objects are derived from the `USBPUMP_USBDI_FUNCTION` type. Both types are in turn derived from the USBPUMP_OBJECT class, and therefore class objects and instance objects can receive USBPUMP_IOCTL messages.

The purpose of the class object is primarily to collect the instances that are controlled by the object, and to control the matching of devices to drivers. Class objects contain pointers to vectors of matching information, which allow matching according to rules given in the USB 2.0/3.0 specification. Instance objects exist to provide context information to the driver code, allowing the driver to operate multiple instances concurrently.

When a new device is detected, a "port object" is assigned to it by USBD. This object is derived from `USBPUMP_USBDI_PORT`. Port objects are the primary API points for USBDI requests; there is a one-to-one correspondence between port objects and active driver instances.

Class objects are matched to device-level ports based either on device class/subclass/protocol or based on vendor ID and product ID. If no class object matches at the device level, then USBD uses standard multi-function techniques to divide the device into logical functions, and assigns one port for each such function. The class drivers are invited to match each function so detected.

After matching, USBD chooses instances from the class' instance pool, and binds them to the port. It then sends the instance a `USBPUMP_IOCTL_EDGE_USBDI_INSTANCE_ARRIVAL` message. The driver code associated with the instance then initiates its internal finite state machine, typically by sending a get-descriptor request to USBDI to get the descriptors it needs to understand the device. Drivers that are internal to USBD (for example the hub and composite device drivers) operate autonomously, calling internal routines within USBD to create and remove device instances as necessary.

Most class drivers, however, provide services to external system components (i.e., to components outside the DataPump environment). MCCI-supplied class drivers provide an object-oriented upper edge, and class-specific APIs. The caller is required to synchronize to the DataPump, if necessary, by posting a DataPump `UCALLBACKCOMPLETION` object (q.v.). Again, this will call a synchronous call into the driver code, this time in response to event from above. Because most embedded operating systems have limited facilities for dynamic device arrival and departure, it is common for the OS class drivers to be active from initialization time. I/Os received from the rest of the system must be completed with errors or delayed if they arrive while the lower (DataPump class driver) instance is not active. In fact, the OS Class Driver is normally implemented so that it's instance data is not held separately from the instance data for the lower driver; this allows USBD to substitute instances, provided that the lower instances support the appropriate abstract APIs.

APIs are provided to allow external drivers and applications to access USBDI. This allows drivers to be written outside the DataPump environment.

Figure 1. Architectural Diagram (by function)

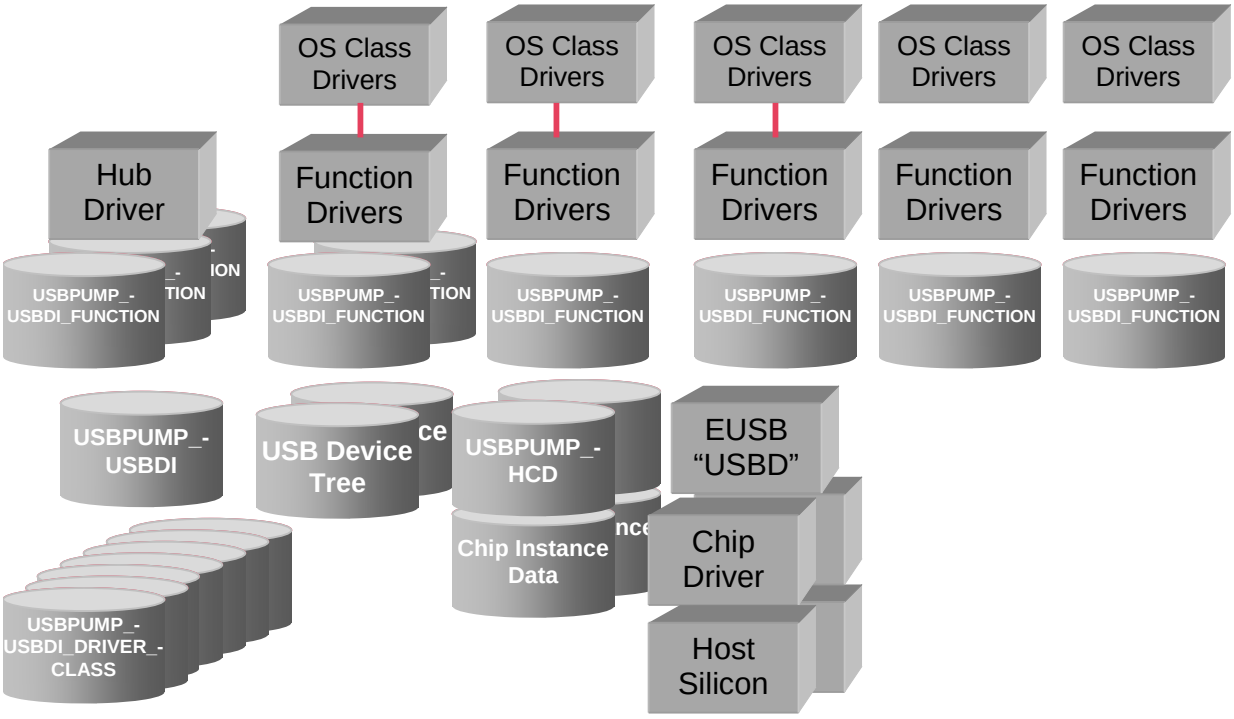
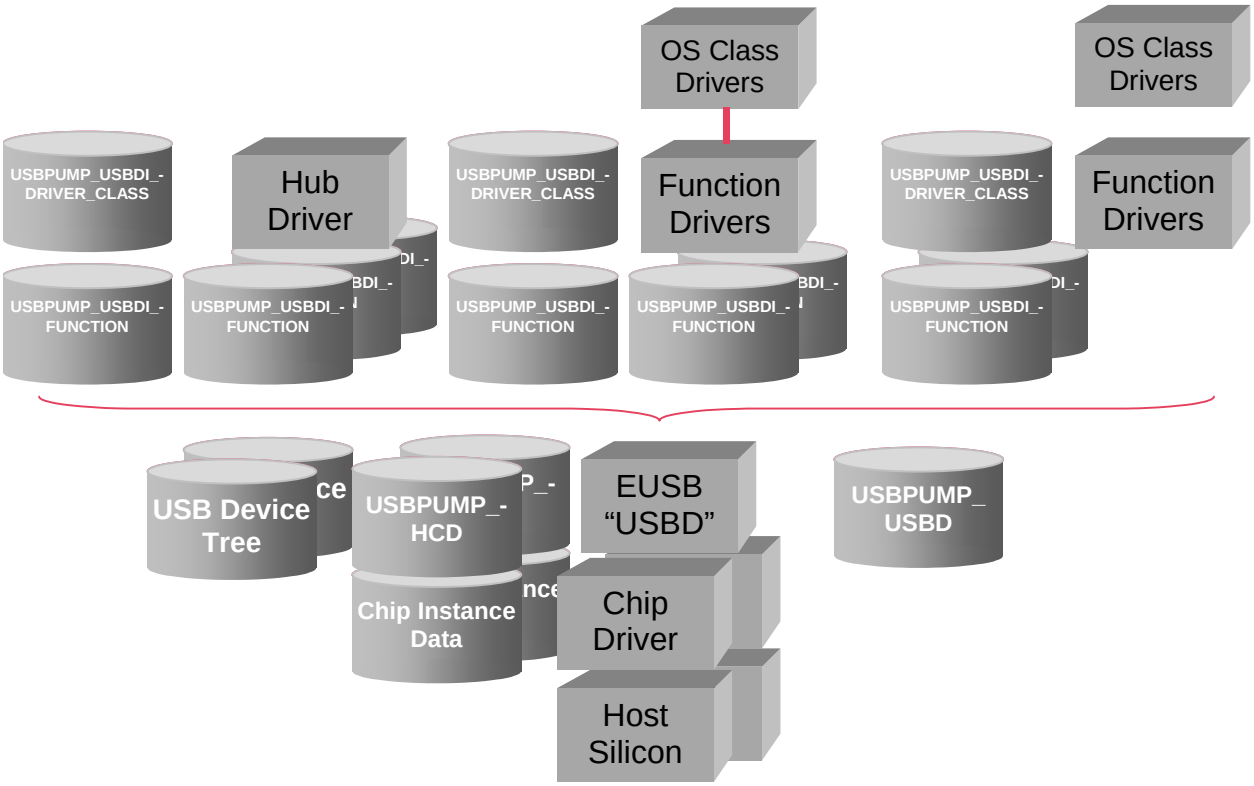


Figure 2. Functional Hierarchy



3 Class Driver Objects

3.1 Naming Conventions

Naming things for USBD is tricky. We currently have the following conventions from [USB2.0/3.0]:

- USBD means the USB Driver
- USBDI means the API to the USB Driver.

The difficulty we have is that it's not clear whether the objects that are owned by USBDI users (but which also are referenced by USBD) should be named "…_USBDI_…" or "…_USBD_…". Some USBD objects, functions and types are internal. On the other hand, coders who are used to Windows will tend to think of USBD, not USBDI.

For now, we are adopting the convention:

Objects that are exposed via the "USBDI" APIs must have names of the form "…_USBDI_…". Objects that are internal to USBD must **not** have names of the form "…_USBDI_…", and further must not be visible when only "usbpump_usbdi.h" is included.

Functions internal to USBD (that would be static if they were all in a single file) may be named "XXX_UsbdJ_…" We do this because of the potential confusion if we named something "XXX_UsbdI_…". Since sometimes we have to create types based on function names, we'd end up with "XXX_USBDI_…" which would not only be confusing, but wrong.

3.2 USBPUMP_USBDI_DRIVER_CLASS

```
typedef union USBPUMP_USBDI_DRIVER_CLASS
   {
   USBPUMP_OBJECT_HEADER                ObjectHeader;
   USBPUMP_OBJECT                       ObjectCast;
   USBPUMP_OBJECT_CLASS_CONTENTS        Class;
   USBPUMP_OBJECT_CLASS                 ClassCast;
   USBPUMP_USBDI_DRIVER_CLASS_CONTENTS  DriverClass;
   };
```

3.3 Creation and Initialization

Class driver objects are derived from USBPUMP_USBDI_DRIVER_CLASS, and are created during system initialization. Each class driver must export a function of type USBPUMP_USBDI_DRIVER_CLASS_INIT_FN:

```
typedef USBPUMP_USBDI_DRIVER_CLASS
USBPUMP_USBDI_DRIVER_CLASS_INIT_FN(
   USTAT *pErrorCode OUT,
```

```
    USBPUMP_OBJECT_HEADER *pParent,
    CONST USBPUMP_USBDI_DRIVER_CLASS_CONFIG *pConfig,
    CONST VOID *pPrivateConfig OPTIONAL
    );
```

The system initialization code (built either manually or using a tool such as [USBRC]) will call this function once, during system initialization. The initialization function creates the driver object for driver, which must be derived from USBPUMP_USBDI_DRIVER_CLASS.

3.3.1 UsbPumpUsbdiLib_CreateDriverClass()

The class driver initialization routine must create the driver class and function objects using the following function:.

```
USBPUMP_USBDI_DRIVER_CLASS *
UsbPumpUsbdiLib_CreateDriverClass(
        USBPUMP_OBJECT_HEADER *pParent,
        CONST USBPUMP_USBDI_DRIVER_CLASS_CONFIG *pConfig,
        CONST USBPUMP_USBDI_DRIVER_CLASS_IMPLEMENTATION *pImplementation
        )
```

pParent should be the parent object header as passed into the driver initialization function. pConfig should be the driver class configuration object, also as passed in to the driver initialization function. The class driver wrapper function adds a pointer to its own (privately defined) implementation object and calls the above routine. The routine allocates a driver class object, and a collection of driver instance (function) objects. It also allocates additional internal data structures that are used by USBDI to provide services for the function driver. This allocation is done at driver creation time in order that we may guarantee that resource limitations in the USB stack will not change the behavior of the class driver when a matching device is plugged in. (Of course, the class driver must also be designed with this in mind; otherwise resource limitations in the class driver may still prevent proper operation.)

In addition to allocating user-visible structures, this function allocates a number of hidden data structures needed for operating the class driver framework within the DataPump. Creating a driver class without the use of this function will most likely result in code that is not portable from version to version of the DataPump.

3.3.2 USBPUMP_USBDI_DRIVER_CLASS_CONFIG

This structure is used to configure a class driver instance. It is intended to allow the system engineer to provide information that's used to tune a class driver for the system's needs.

The config object may be initialized either at compile time or at runtime.

For compile-time initialization, use:

```
CONST USBPUMP_USBDI_DRIVER_CLASS_CONFIG MyConfig =
    USBPUMP_USBDI_DRIVER_CLASS_CONFIG_INIT_V1(
        CONST USBPUMP_USBDI_INIT_MATCH_LIST *pInitMatchList,
        CONST TEXT *pClassName,
        CONST TEXT *pFunctionName,
        BYTES NumInstances
        );
```

For run-time initialization of the structure, use:

```
VOID
USBPUMP_USBDI_DRIVER_CLASS_CONFIG_SETUP_V1(
        USBPUMP_USBDI_DRIVER_CLASS_CONFIG *pConfig,
        CONST USBPUMP_USBDI_INIT_MATCH_LIST *pInitMatchList,
        CONST TEXT *pClassName,
        CONST TEXT *pFunctionName,
        BYTES NumInstances
        );
```

`pInitMatchList` points to a data structure that describes the matching IDs It is used to create the appropriate `USBPUMP_USBDI_MATCH_LIST_HEADERS` (section 3.5).

The string given by `pClassName`, if not `NULL`, is used to override the driver class name provided by the driver's implementation. If it is `NULL`, the class driver's default is used. This name should be created using the `USBPUMP_USBDI_DRIVER_CLASS_NAME()` macro, in order to match DataPump naming conventions.

The string given by `pFunctionName`, if not `NULL`, is used to override the function instance objects name. If it is `NULL`, the class driver's default is used. This name should be created using the `USBPUMP_USBDI_FUNCTION_NAME()` macro, in order to match DataPump naming conventions.

`NumInstances` specifies how many instances should be created for this driver. If zero, the driver class will not be created.

3.3.3 USBPUMP_USBDI_DRIVER_CLASS_IMPLEMENTATION

This structure is used to describe a driver class to `UsbPumpUsbdiLib_CreateDriver-Class()`. The structure is normally initialized at compile time using the following macro:

```
CONST USBPUMP_USBDI_DRIVER_CLASS_IMPLEMENTATION MyClassImplementation =
    USBPUMP_USBDI_DRIVER_CLASS_IMPLEMENTATION_INIT_V1(
        BYTES sizeClass,
        BYTES sizeFunction,
        CONST TEXT *pClassName,
        CONST TEXT *pFunctionName,
        USBPUMP_OBJECT_CLASS_IOCTL_FN *pClassIoctlFn,
        USBPUMP_OBJECT_IOCTL_FN *pFunctionIoctlFn,
        BYTES pipesPerInstance,
```

```
        BYTES sizeDriverPool,
        BYTES sizeInstancePool,
        BYTES sizeManagementPool
    );
```

`sizeClass`	The desired size of the class object, in bytes. This value must be ≥ `sizeof(USBPUMP_USBDI_DRIVER_CLASS)`.
`sizeFunction`	The desired size of each function object, in bytes. This value must be ≥ `sizeof(USBPUMP_USBDI_FUNCTION)`.
`pClassName`	The default name of the class object.
`pFunctionName`	The default name for the function objects.
`pClassIoctlFn`	The function to be called for providing class-specific extensions to the class object IOCTL behavior.
`pFunctionIoctlFn`	The function to be called for providing driver-specific extensions to the function object IOCTL behavior.
`pipesPerInstance`	The maximum number of pipes expected to be allocated by a given instance of this class driver. This is the maximum over all configurations. If the class driver only will operate the device in a single configuration, it can minimize the number of pipes by only submitting the configuration it wishes to use to DEFINE_CONFIG. Similarly, if the class driver will only operate the device with a subset of the available alternate settings, it can minimize the number of pipes by only submitting those alternate settings it wishes to use.
`sizeDriverPool`	Number of bytes to allocate for the Driver Pool.
`sizeInstancePool`	Number of bytes to allocate for the Instance Pool.
`sizeManagementPool`	Number of bytes to allocate for the Management Pool.

3.3.4 USBPUMP_USBDI_DRIVER_CLASS_TAG

USBPUMP_USBDI_DRIVER_CLASS_TAG is a macro defined as UHIL_MEMTAG('U', 'C', 'l', 's')

The value of UHIL_MEMTAG depends upon the endianess. If the Endian is Little Endian, UHIL_MEMTAG(a, b, c, d) is defined as (((d) << 24) | ((c) << 16) | ((b) << 8) | (a)). If the Endian is PDP, UHIL_MEMTAG(a, b, c, d) is defined as (((b) << 24) | ((a) << 16) | ((d) << 8) | (c)) and if the Endian is Big Endian, UHIL_MEMTAG(a, b, c, d) is defined as (((a) << 24) | ((b) << 16) | ((c) << 8) | (d))

3.3.5 USBPUMP_USBDI_DRIVER_CLASS_NAME()

The macro USBPUMP_USBDI_DRIVER_CLASS_NAME("name") generates a string consisting of "name" followed by ".driver.usbdi.mcci.com". Therefore, the appropriate way to match all driver class is to match the string USBPUMP_USBDI_DRIVER_CLASS_NAME("*").

3.4 USBPUMP_USBDI_FUNCTION

Instance objects are derived from USBPUMP_USBDI_FUNCTION objects. Represents a single function-driver instance to USBDI.

Each class driver owns a collection of USBPUMP_USBDI_FUNCTION objects; these objects are created during class-driver initialization, and are dynamically connected to USBPUMP_USBDI_PORT objects based on class type matching.

These function objects are DataPump extensible objects, and so this type can be thought of as an abstract view of the concrete function object, exported by the concrete class driver.

These function objects are furthermore part of the DataPump class system, so they can be located using UsbPumpObject_EnumerateMatchingNames(). Names are normally constructed at compile time using the USBPUMP_USBDI_FUNCTION_NAME() macro.

3.5 Device Matching

Class driver objects contain a pointer to a list of `USBPUMP_USBDI_MATCH_LIST_HEADER` entries. Each such entry consists of a header followed by a vector or one or more pointers to strings. Each string is used as a pattern argument to `UsbPumpLib_MatchPattern`, and is compared to the device ID string generated for the instance by USBDI.

These entries are normally prepared from static `USBPUMP_USBDI_INIT_MATCH_LIST` objects.

For each device, USBDI will generate one of the following strings:

```
vid=####/####;r=####;dc=##/##/##;

vid=####/####;r=####;if=##;ic=##/##/##;

vid=####/####;r=####;dc=##/##/##;if=##;ic=##/##/##;

vid=####/####;r=####;dc=##/##/##;if=##;ic=##/##/##;ig={########-####-####-####-
############}
```

3.5.1 USBPUMP_USBDI_MATCH_LIST_HEADER

```
typedef struct USBPUMP_USBDI_MATCH_LIST_HEADER
   {
   USBPUMP_USBDI_MATCH_LIST_HEADER  *pNext;
   USBPUMP_USBDI_MATCH_LIST_HEADER  *pLast;
```

```
BYTES                           nPatterns;
USBPUMP_USBDI_MATCH_LIST_ENTRY  Entry[1];
};
```

3.5.2 USBPUMP_USBDI_MATCH_LIST_ENTRY

```
typedef struct USBPUMP_USBDI_MATCH_LIST_ENTRY
   {
   CONST TEXT *                 pPattern;
   UINT16                       nPattern;
   USBPUMP_USBDI_MATCH_PRIORITY uPriority;
   };
```

3.5.3 Match Priorities

Because drivers may be loaded in any sequence and a specific match may be seen later in the driver sequence than a generic match, we need a way to prioritize matches. Match priorities are used to do this, and should be assigned using the following symbols:

Table 1. Match Priorities

Priority Name	**Meaning**
USBPUMP_USBDI_PRIORITY_VIDPIDREV	Match is made at the VID/PID/Revision level
USBPUMP_USBDI_PRIORITY_VIDPID	Match is made at the VID/PID level
USBPUMP_USBDI_PRIORITY_VIDPIDREV_IF	Match is made at the VID/PID/REV and interface-number level (this is for a multi-function device)
USBPUMP_USBDI_PRIORITY_VIDPID_IF	Match is made at the VID/PID and interface-number level
USBPUMP_USBDI_PRIORITY_DEV_CSP	Match is made at the device class/subclass/protocol level
USBPUMP_USBDI_PRIORITY_DEV_CS	Match is made at the device class/subclass level
USBPUMP_USBDI_PRIORITY_DEV_C	Match is made at the device class level.
USBPUMP_USBDI_PRIORITY_VIDPIDREV_FN_CSP	Match is made at the device VID/PID/REV level, to a function of the device using class/subclass/protocol.
USBPUMP_USBDI_PRIORITY_VIDPID_FN_CSP	Match is made at the device VID/PID level, to a function of the device using class/subclass/protocol.
USBPUMP_USBDI_PRIORITY_VIDPIDREV_FN_CS	Match is made at the device VID/PID/REV level, to a function of the device using class/subclass.
USBPUMP_USBDI_PRIORITY_VIDPID_FN_CS	Match is made at the device VID/PID level, to a function

Priority Name	Meaning
	of the device using class/subclass.
USBPUMP_USBDI_PRIORITY_VIDPIDREV_FN_C	Match is made at the device VID/PID/REV level, to a function of the device using class.
USBPUMP_USBDI_PRIORITY_VIDPID_FN_C	Match is made at the device VID/PID level, to a function of the device using class.
USBPUMP_USBDI_PRIORITY_FN_GUID	Match is made at the function level, using a GUID as provided (for example) by CDC WMC MDLM interfaces.
USBPUMP_USBDI_PRIORITY_FN_CSP	Match is made at the function level, using class, subclass, protocol.
USBPUMP_USBDI_PRIORITY_FN_CS	Match is made at the function level, using class and subclass.
USBPUMP_USBDI_PRIORITY_FN_C	Match is made at the function level, using class
USBPUMP_USBDI_PRIORITY_WEAK	Match is very weak, but better than zero. This is useful for wildcards.

3.5.4 USBPUMP_USBDI_MATCH_ID_LENGTH

This symbol is defined for convenience, as it may be used for the size of a buffer for the maximum length match pattern that USBDI will generate.

4 USBDI URBs

Class drivers communicate with their devices using USBDI Requests, which are sent using USBDI Request Blocks (called URBs, type `USBPUMP_URB`s). URBs are variable-length message packets. The first part of the URB is a header that is common for every request. The layout of this header is determined by USBDI. Based on the request, additional parameters may be included; these appear after the header. As a consequence, the USBDI-defined portion of the URB will vary in size, based on the particular request. In addition, as a convenience for the HCD, some bytes in each packet are reserved for use by the HCD. The number of bytes is fixed for any given HCD, but may vary from HCD to HCD, and from system to system. Class drivers get the size of the HCD fixed area by saving the value provided in the `USBPUMP_IOCTL_EDGE_USBDI_INSTANCE_ARRIVAL_ARG` parameter. Some URBs require more data than others in the USBDI; however, the USBDI-specified portion of the Some requests also require an additional variable-length area to be passed in the URB; these requirements must also be factored into the allocation of the URB. All URBs are self-sizing; and the constructor of the URB is allowed to place the HCD fixed area at any convenient offset within the fixed tail of the URB provided that normal alignment constraints are met.

4.1 Preparing and Submitting a URB

The basic process is the same for all requests.

1. Determine the size of the request. This is done by adding the size needed by the URB variant to the size reported by `USBPUMP_IOCTL_USBDI_PORT_GET_URB_SIZE`.

2. Fill in the request, except for `pPort`, `pDoneFn` and `pDoneInfo`.

3. Determine a callback routine. The signature of this routine should be USBPUMP_URB_DONE_FN (See section 5.1.1.1).

   Even if the request is badly malformed, the system will be able to call the callback routine with correct results.

4. Call the URB submission entry point in the parent `USBPUMP_USBDI_PORT` object:. If `pFunction` points to the function driver's function object, derived from `USBPUMP_USBDI_FUNCTION`, then the following code would be used to submit the request.

   ```
   (*pFunction->Function.PortInfo.pPort->Port.pMethods->pSubmitRequest)(
       pFunction->Function.PortInfo.pPort,
       pUrb,
       pDoneFn,
       pDoneInfo
       )
   ```

5. Do other work (or simply return to the DataPump event loop).

6. When the completion routine is called, check the status (passed as a parameter, and also available at `pUrb->UrbHdr.Status`), re-evaluate the software state of the instance, and do any needed follow-on work.

For convenience, URBs have a timeout field. If the timeout field is non-zero, USBD will automatically cancel the URB after a certain period of time if I/O fails to progress within the HCD queues.

4.2 Cancelling a URB

If the default timeout mechanism is not sufficient, the function driver may have to cancel an outstanding URB.

To cancel a URB, call the URB cancellation method `Port.pMethods->pCancelRequest` of the relevant `USBPUMP_USBDI_PORT` object.

It's normally invoked as shown below, assuming that `pFunction` points to this function driver's derived `USBDI_FUNCTION` object:

```
(*pFunction->Function.pPort->Port.pMethods->pCancelRequest)(
        pFunction->Function.pPort,
        pRequest
        );
```

4.3 Issuing Standard USB Chapter 9 Requests

Unlike many USBDI implementations, the DataPump USBDI does not have separate requests for performing operations that result in issuing standard requests. Instead, the client creates a standard CONTROL_IN or CONTROL_OUT requests containing the standard request. The DataPump USBD detects the standard request and automatically performs the required bookkeeping operations. For reference, we list the operations often found in other USBDI implementations, and the equivalent operations in the DataPump USBDI in Table 2.

One consequence of this architecture is that the default pipe for a device is a full-fledged pipe, and is not distinct at this API level from any other kind of pipe.

Table 2. Standard Requests as Implemented in the DataPump USBDI

Operation	Usual Implementation	DataPump USBDI Equivalent
SELECT_CONFIGURATION	A special request, accompanied by a list of pipe and bandwidth specifications	First, use `USBPUMP_URB_RQ_DEFINE_CONFIG`, then send `USBPUMP_URB_RQ_CONTROL_OUT` containing a SET_CONFIGURATION request to the default pipe for the device. If the device was previously configured, and any requests are pending to any of the pipes in the old configuration, then this request will be rejected. This is true even if the configuration is not being changed. All pipes in the new configuration will be reset. This is true even if the configuration is not being changed.
SELECT_INTERFACE	A special request, accompanied by a list of pipe and bandwidth specifications	Use `USBPUMP_URB_RQ_DEFINE_CONFIG` to establish the device configuration parameters in USBDI, then send `USBPUMP_URB_RQ_CONTROL_OUT` containing a SET_CONFIGURATION request to the default pipe for the device to select the configuration. Then send `USBPUMP_URB_RQ_CONTROL_OUT` containing a properly-formatted SET_INTERFACE request to the default pipe If any requests are pending to any of the pipes in the old alternate setting, then this request will be rejected. All pipes in the new alternate setting will be reset
GET_DESCRIPTOR	A special request, taking specific	Use `USBPUMP_URB_RQ_CONTROL_IN` containing a

Operation	Usual Implementation	DataPump USBDI Equivalent
	arguments	GET_DESCRIPTOR request
SET_DESCRIPTOR	A special request, taking specific arguments.	Use USBPUMP_URB_RQ_CONTROL_OUT containing a SET_DESCRIPTOR request
SET_FEATURE	A special request, taking specific arguments	Use USBPUMP_URB_RQ_CONTROL_OUT containing a SET_FEATURE request
CLEAR_FEATURE	A special request, taking specific arguments	Use USBPUMP_URB_RQ_CONTROL_OUT containing a CLEAR_FEATURE request
GET_STATUS	A special request, taking specific arguments	Use USBPUMP_URB_RQ_CONTROL_IN containing a GET_FEATURE request
GET_CONFIGURATION	A special request, taking specific arguments	Use USBPUMP_URB_RQ_CONTROL_IN containing a GET_CONFIGURATION request
GET_INTERFACE	A special request, taking specific arguments	Use USBPUMP_URB_RQ_CONTROL_IN containing a GET_INTERFACE request
ENABLE_REMOTE_-WAKEUP	A separate request, possibly handled via the SET_FEATURE API	Use USBPUMP_URB_RQ_CONTROL_OUT containing a SET_FEATURE request

5 USBPUMP_USBDI_PORT

This is the basic instance object for an attachment point for a function driver to USBDI. A port has the following externally-visible entries.

```
USBPUMP_USBDI_PORT_SUBMIT_REQUEST_FN *     Port.pMethods->pSubmitRequest;
         A pointer to the URB submission function.

USBPUMP_USBDI_PORT_CANCEL_REQUEST_FN *     Port.pMethods->pCancelRequest;
         A pointer to the URB cancellation function.
```

5.1 Port Method Functions

5.1.1 Submit Request

```
VOID (*Port.pMethods->pSubmitRequest)(
          USBPUMP_USBDI_PORT *pPort,
          USBPUMP_URB *pUrb,
          USBPUMP_URB_DONE_FN *pDoneFn,
          VOID *pDoneInfo
          );
```

Submits a request to USBDI for processing. The function `pDoneFn`, if non-`NULL`, will always be called when USBDI finishes processing the URB. It is an error for `pDoneFn` to be `NULL`, so this method checks `pDoneFn` first, and immediately returns without doing any further processing if it is `NULL`.

`pPort`, `pDoneFn` and `pDoneInfo` correspond to fields in the URB, so the reader may wonder why these are not simply passed by the client in the URB. The reason is to ensure consistent operation in case the URB pointed to by pUrb is invalid. Without having these parameters separate, we could not guarantee that `pDoneFn` would be called if (for example) `pUrb` is `NULL`. `pSubmitRequest()` validates the URB header before storing `pPort`, `pDoneFn` and `pDoneInfo` into the URB. The values passed in the URB itself for these three parameters are ignored and are overwritten by `pSubmitRequest()`.

5.1.1.1 Request Completion Functions (`USBPUMP_URB_DONE_FN`)

```
typedef VOID
USBPUMP_URB_DONE_FN(
          USBPUMP_USBDI_PORT *pPort,
          USBPUMP_URB *pUrb,
          VOID *pDoneInfo,
          ARG_USTAT Status
          );
```

This function is called when USBDI is finished processing a URB. The arguments `pPort`, `pUrb`, and `pDoneInfo` are the same values that were passed when the request was submitted using **`Port.pMethods->pSubmitRequest`**. The argument `Status` is the completion status of the request.

Status is normally available in `pUrb->UrbHdr.Status`, so it may be wondered why we have this parameter. The reason is for consistency in the case of invalid parameters. For example, the Submit Request operation will fail

if `pUrb` is `NULL`. We pass `Status` separately so that the completion routine can always have a valid status code to work with, even if the request was rejected because `pUrb` was invalid.

5.1.2 Cancel Reqeust

```
VOID (*Port.pMethods->pCancelRequest)(
          USBPUMP_USBDI_PORT *pPort,
          USBPUMP_URB *pUrb
          );
```

Initiate cancellation of a pending request. Note that the request might not immediately complete, depending on its current state; and in fact, cancellation might have no effect at all. But generally speaking, USBDI will attempt to finish up the URB as quickly as it can. a URB that fails because it was cancelled will have status USTAT_KILL.

6 Using URBs

6.1 The URB Union

Different URB requests require different arguments and therefore use different input structures. Following more or less modern practice, a USBPUMP_URB object is defined which collects all the variants into a single collection.

Table 3. The USBPUMP_URB Union

typedef union **{**	**Description**
USBPUMP_URB_HDR **UrbHdr**;	The common URB header
USBPUMP_URB_DEVICE_CONTROL **UrbDeviceControl**;	The URB used for device control operations (RESET_PORT, REENUMERATE_PORT)
USBPUMP_URB_GET_DEVICE_INFO **UrbGetDeviceInfo**;	The URB used for obtaining information about the device. Normally used only at enumeration time.
USBPUMP_URB_GET_FRAME **UrbGetFrame**;	The URB used for getting the current frame number
USBPUMP_URB_PIPE_CONTROL **UrbPipeControl**;	The URB used for pipe control operations (ABORT_PIPE, RESET_PIPE)
USBPUMP_URB_PIPE_INOUT **UrbPipeInOut;**	The URB pipe to read and write

typedef union **{**	**Description**
USBPUMP_URB_BULKINT_IN **UrbBulkIntIn;**	The URB used for reading from bulk or interrupt pipes
USBPUMP_URB_BULKINT_OUT **UrbBulkIntOut;**	The URB used for writing to bulk or interrupt pipes.
USBPUMP_URB_BULKINT_INOUT **UrbBulkIntInOut;**	The URB used for read and write of bulk or interrupt pipes
USBPUMP_URB_CONTROL_IN **UrbControlIn;**	The URB used for reading from control pipes (including the default pipe).
USBPUMP_URB_CONTROL_OUT **UrbControlOut;**	The URB used for writing to control pipes (including the default pipe).
USBPUMP_URB_CONTROL_INOUT **UrbControlInOut;**	The URB used for read and write of control pipes
USBPUMP_URB_ISOCH_IN **UrbIsochIn;**	The URB used for reading from Isochronous pipes
USBPUMP_URB_ISOCH_OUT **UrbIsochOut;**	The URB used for writing to Isochronous pipes.
USBPUMP_URB_ISOCH_INOUT **UrbIsochInOut;**	The URB used for read and write of Isochronous pipes
USBPUMP_URB_DEFINE_CONFIG **UrbDefineConfig;**	The URB used to tell USBD and the host controller about the desired set of possible operations for this device. This URB does not result in any I/O to the target device, but rather causes USBD and the HCD to allocate internal resources to ensure that any of the defined configurations may be selected. Note that the success of this request does not imply that the device can be configured – there might not be power or bandwidth available at the time the configuration is selected.
USBPUMP_URB_SUGGEST_CONFIG **UrbSuggestConfig;**	The URB used to ask USBD to suggest an appropriate configuration tree for the device or function. Normally USBDI uses the device's descriptors, but the caller may suggest a substitute set of descriptors to be parsed.
USBPUMP_URG_GET_PORT_STATUS **UrbGetPortStatus;**	The URB used for obtaining status from the port.
USBPUMP_URB_BULKINT_STREAM_IN **UrbBulkIntStreamIn;**	The URB used for reading from bulk interrupt or stream pipes

typedef union **{**	**Description**
USBPUMP_URB_BULKINT_STREAM_OUT **UrbBulkIntStreamOut;**	The URB used for writing to bulk interrupt or steam pipes
USBPUMP_URB_BULKINT_STREAM_INOUT **UrbBulkIntStreamInOut;**	The URB used for read and write of bulk or interrupt or steam pipes
USBPUMP_URB_RESET_PIPE **UrbResetPipe;**	The URB reset pipe
} **USBPUMP_URB**;	

6.2 Request Codes

A summary of the URB request codes is shown in Table 4.

Table 4. URB Request Codes

Name	Description
USBPUMP_URB_RQ_ABORT_PIPE	Abort pending operations on a pipe
USBPUMP_URB_RQ_BULKINT_IN	Read from a bulk or interrupt pipe
USBPUMP_URB_RQ_BULKINT_OUT	Write to a bulk or interrupt pipe
USBPUMP_URB_RQ_CONTROL_IN	Read from a control pipe
USBPUMP_URB_RQ_CONTROL_OUT	Write to a control pipe
USBPUMP_URB_RQ_DEFINE_CONFIG	Define device configuration
USBPUMP_URB_RQ_GET_DEVICE_INFO	Get information about device
USBPUMP_URB_RQ_GET_FRAME	Get current frame
USBPUMP_URB_RQ_GET_PORT_STATUS	Get port status
USBPUMP_URB_RQ_ISOCH_IN	Read from an isochronous pipe
USBPUMP_URB_RQ_ISOCH_OUT	Write to an isochronous pipe
USBPUMP_URB_RQ_REENUMERATE_PORT	Reenumerate the port
USBPUMP_URB_RQ_RESET_PIPE	Reset data toggle and clear halt condition on a pipe

Name	Description
USBPUMP_URB_RQ_RESET_PORT	Reset the port (and restore the current configuration)
USBPUMP_URB_RQ_SUGGEST_CONFIG	Build suggested configuration for the associated device
USBPUMP_URB_RQ_BULKINT_STREAM_IN	Read from a bulk interrupt or stream pipe
USBPUMP_URB_RQ_BULKINT_STREAM_OUT	Write to a bulk interrupt or stream pipe

6.3 Common URB request header

All URBs begin with the following fields

Table 5. URB header fields

Name	Description
USBPUMP_URB_LENGTH Length;	Length of the URB structure in bytes, including any HCD-specific fields.
USBPUMP_URB_CODE Request;	The URB request code.
USTAT Status;	The URB status. Set to zero (USTAT_BUSY) when the URB is passed down for processing, and set non-zero on completion. While the URB is owned by USBDI, this should be treated as opaque information.
UINT8 Refcount;	The URB reference counter.
UINT8 InternalFlags;	For internal use by USBDI only. Initialize to zero. In other words, clients of USBDI should not use this field.
UNIT8 InternalStatus;	For internal use by USBDI only. Initialize to zero. In other words, clients of USBDI should not use this field.
USBPUMP_URB *pNext;	General-purpose link field. While the URB is owned by USBDI, this should be treated as opaque information.
USBPUMP_URB *pLast;	General-purpose link field. While the URB is owned by USBDI, this should be treated as opaque information.
USBPUMP_URB_DONE_FN *pDoneFn;	Completion function – initialized by USBDI from the pDoneFn parameter passed when the URB is submitted.
VOID *pDoneInfo	Information for the completion function – initialized by USBDI from the pDoneInfo parameter passed in when the URB is submitted.

Name	Description
USBPUMP_USBDI_PORT *pUsbdPort;	Pointer to the parent port object – initialized by USBDI from the pPort parameter passed in when the URB is submitted.
USBPUMP_USBDI_PORT_KEY PortKey;	The authentication key. This key is issued to the function instance when the device arrives, and it's changed when the device departs. Any requests issued must be accompanied by the port key; if the port key doesn't match, then the request is stale and is rejected.
USBPUMP_URB_CANCEL_FN *pCancelFn;	Pointer to URB cancellation function. Initialized to NULL by USBDI when the URB is passed down submitted for processing. While the URB is owned by USBDI, this should be treated as opaque information. While the URB is owned by USBDI, this should be treated as opaque information. In other words, clients of USBDI should not use this field.
VOID *pCancelInfo;	Pointer to cancellation information. While the URB is owned by USBDI, this should be treated as opaque information. In other words, clients of USBDI should not use this field.
USBPUMP_USBDI_HCD_INDEX HcdRequestIndex;	Byte index to the bytes reserved for the HCD request block within this URB. The size of the request block is assumed to be the same as was specified in the arguments to the USBPUMP_IOCTL_EDGE_USBDI_INSTANCE_ARRIVAL notification.
USBPUMP_USBDI_TIMEOUT HcdTimeout;	Request timeout in ms (when request is submitted to the HCD).

Transfer URBs all have a field named TransferFlags. The bits are summarized in Table 6. Refer to the particular transfer for information about the meaning of the flag, if any, when used on a particular request.

Table 6. Transfer flags

Name	Description
USBPUMP_URB_TRANSFER_FLAG_ASAP	If set, the transfer is to begin as soon as possible.
USBPUMP_URB_TRANSFER_FLAG_LINKED	If set, the URB is the first of a set of linked URBs. pUrb->UrbHdr.pNext points to the head, and pUrb->UrbHdr.pLast points to the tail of the circular queue of URBs.
USBPUMP_URB_TRANSFER_FLAG_SHORT_OK	If set, a short transfer is OK
USBPUMP_URB_TRANSFER_FLAG_POST_BREAK	If set, and if the transfer would not end with a short packet, then USBD will ensure that a ZLP is sent as part of the

Name	Description
	transfer

6.3.1 Isochronous Transfer Packet Descriptors

Isochronous transfers are packet-oriented, rather than stream oriented. Therefore, they require that a second buffer be supplied, containing an array of USBPUMP_ISOCH_PACKET_DESCR elements. These elements effectively divide the client's transfer buffer into packet buffers, and provide space for USBD to record the status of the transfers for each packet. See Figure 3 for a picture of the relationships. The contents of a single descriptor are explained in Table 7.

Figure 3. Isochronous Packet Descriptors and Transfer Buffers

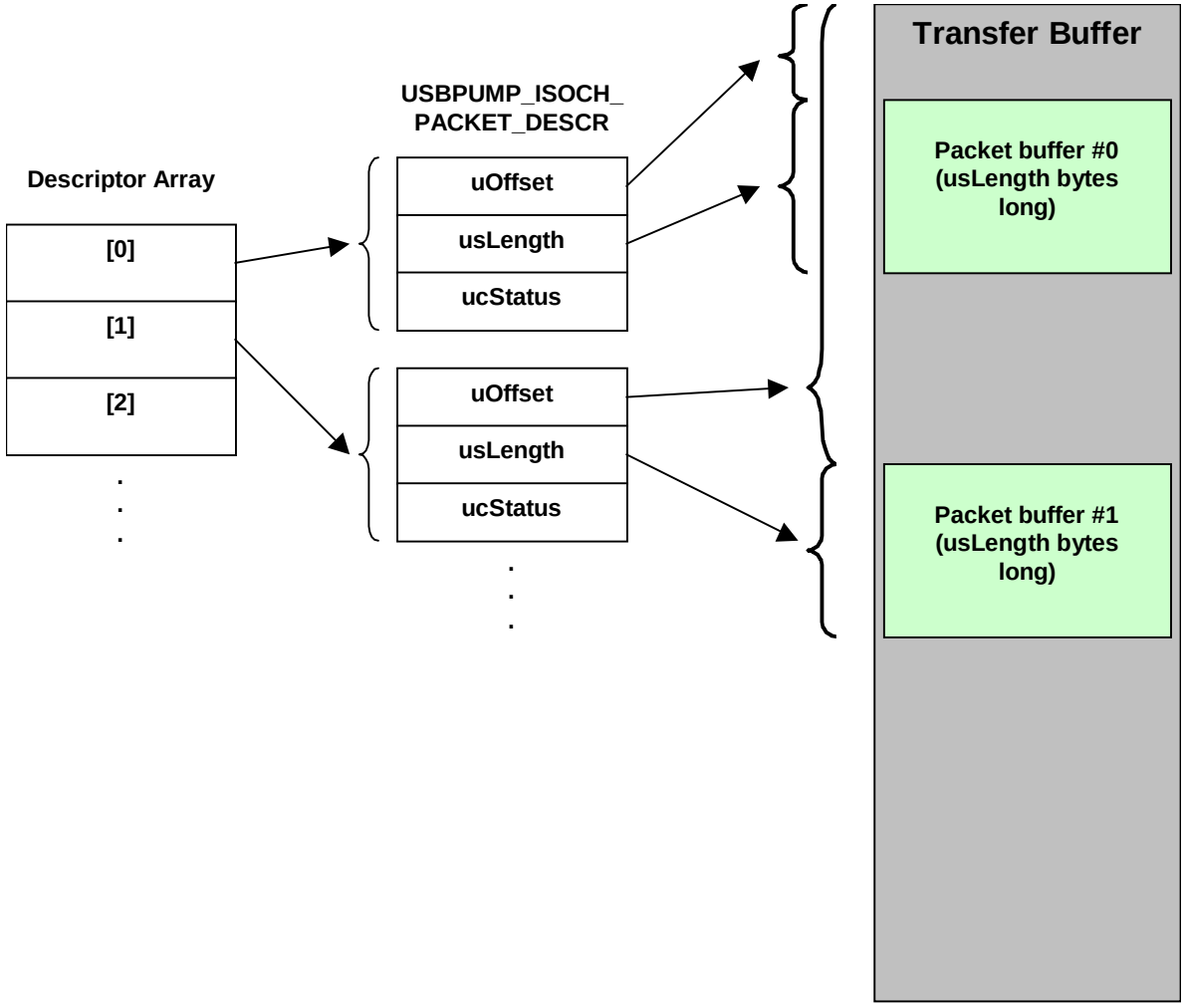


Table 7. The `USBPUMP_ISOCH_PACKET_DESCR` Structure

`typedef struct` `{`	Name

typedef struct **{**	**Name**
BYTES **uOffset**;	Offset to the packet within the transfer buffer
UINT16 **usLength**;	Input: length of the packet buffer. Output: actual size of the packet.
UINT8 **ucStatus**;	The USTAT status for this packet's transaction.
UINT8 **ucSpare**;	Reserved for future use.
} **USBPUMP_ISOCH_PACKET_DESCR**;	

6.4 Extra Space In URBs

USBDI's URBs are structured as a wrapper containing the URB, an HCD request, some number of extra bytes allocated by the port handler for management purposes, and optionally some number of bytes for use by the allocator of the URB. The memory layout is shown in figure 5 .

The areas of memory in the URB are, sequentially:

1. The USBPUMP_URB per se – this is the structure that contains the information prepared by the caller. We reduce

2. Extra bytes for use by the caller (as determined by the number of additional bytes added to the `UrbExtraSize` value passed in at enumeration time).

3. Extra bytes for the port layers: bytes from `pUrb + pUrb->UrbHeader.Length - UrbExtraSize` to `pUrb + pUrb->UrbHeader.Length – HcdRequestSize`.

4. The HCD request field, located at `pUrb + pUrb->UrbHeader.Length – HcdRequestSize`.

The USBDI dispatcher ensures that all these fields are non-overlapping and are correctly aligned, and rejects any request that is doesn't provide enough space.

There are several things to note.

The client who prepares the URB is free to size the request portion of the URB as tightly or loosely as makes sense to the client. In other words, all URBs may be sized with a fixed size block representing the specific request, followed by an HCD request and port-specific data at fixed offsets within the request block. Alternately, the port may choose to allocate URBs with only enough room for the specific request variant, followed immediately by the HCD request block and port-specific data. Finally, the client may choose to leave extra space between the URB and the first byte of the port layer; the client may use this space for any purpose.

6.5 USBPUMP_URB_RQ_ABORT_PIPE

This request causes USBDI to abort all requests that are pending for the specified pipe.

This request uses the USBPUMP_URB_PIPE_CONTROL variant of the URB, named `UrbPipeControl`. The additional fields are shown in Table 8.

Table 8. Request-specific fields for `USBPUMP_URB_PIPE_CONTROL`

Name	Description
`USBPUMP_USBDI_PIPE_HANDLE hPipe;`	Specifies the target pipe – this is obtained from the result of `USBPUMP_URB_RQ_DEFINE_CONFIG`.

No guarantees are made about the completion order of any cancelled URBs, relative to the completion of this URB.

This URB is prepared using the function UsbPumpUrb_PrepareControlIn()

6.6 USBPUMP_URB_RQ_BULKINT_IN

This request causes USBDI to read data from a specified pipe. The pipe may be either a bulk or interrupt pipe; the semantics are similar enough that we feel it's convenient to allow drivers to ignore the distinctions after the pipe has been identified and opened.

This request takes a `USBPUMP_URB_BULKINT_IN` structure as its parameter. The additional fields are shown in Table 9.

Table 9. Additional fields for USBPUMP_URB_BULKINT_IN

Name	Description
`USBPUMP_USBDI_PIPE_HANDLE hPipe;`	Specifies the target pipe – this is obtained from the result of `USBPUMP_URB_RQ_DEFINE_CONFIG`.
`UINT32 TransferFlags;`	**The transfer flags. See** Table 10 for the applicable flags
`VOID *pBuffer;`	Pointer to base of user buffer.
`BYTES nBuffer;`	Number of bytes in the buffer.
`BYTES nActual;`	Output: the number of bytes actually transferred to the buffer.

Table 10. Transfer flags for Bulk and Interrupt Reads

Name	Description
USBPUMP_URB_TRANSFER_FLAG_SHORT_OK	If set, a short packet (with size less than wMaxPacketSize for this pipe) is acceptable (although it will cause the transfer to be retired). If clear, a short packet will cause the transfer to be retired with an error.
USBPUMP_URB_TRANSFER_FLAG_LINKED	If set, the URB is the first of a set of linked URBs. pUrb->UrbHdr.pNext points to the head, and pUrb->UrbHdr.pLast points to the tail of the circular queue of URBs.

6.7 USBPUMP_URB_RQ_BULKINT_OUT

This request causes USBDI to write data to a specified pipe. The pipe may be either a bulk or interrupt pipe; the semantics are similar enough that we feel it's convenient to allow drivers to ignore the distinctions after the pipe has been identified and opened.

This request takes a USBPUMP_URB_BULKINT_OUT structure as its parameter. The additional fields are shown in Table 11.

Table 11. Additional fields for USBPUMP_URB_BULKINT_OUT

Name	Description
USBPUMP_USBDI_PIPE_HANDLE hPipe;	Specifies the target pipe – this is obtained from the result of USBPUMP_URB_RQ_DEFINE_CONFIG.
UINT32 TransferFlags;	The transfer flags. See Table 12 for the applicable flags
CONST VOID *pBuffer;	Pointer to base of user buffer.
BYTES nBuffer;	Number of bytes in the buffer.
BYTES nActual;	Output: the number of bytes actually transferred to the buffer.

Table 12. Transfer flags for Bulk and Interrupt Writes

Name	Description
USBPUMP_URB_TRANSFER_FLAG_LINKED	If set, the URB is the first of a set of linked URBs. pUrb->UrbHdr.pNext points to the head, and pUrb->UrbHdr.pLast points to the tail of the circular queue of URBs.

Name	Description
USBPUMP_URB_TRANSFER_FLAG_POST_BREAK	If set, and if the transfer would not end with a short packet, then USBD will ensure that a ZLP is sent as part of the transfer

6.8 USBPUMP_URB_RQ_CONTROL_IN

This request causes USBDI to issue a control-IN transfer to a specified control pipe (possibly the default pipe). The SETUP packet is transported within the URB. If the request targets the default pipe, and is a standard request, additional processing is done as needed to ensure that the system-wide invariants with respect to the USB bus are preserved. If the function issuing a request is part of a composite device, then the request may be emulated or blocked as necessary.

This request takes a USBPUMP_URB_CONTROL_IN structure as its parameter. The additional fields are shown in Table 13.

According to the control transfer protocol outlined in section 8.5.3 of [USBCORE], if `pUrb->UrbControlIn.nBuffer` is zero, then the sequence of transactions on the bus shall be SETUP / IN(ZLP), where the IN(ZLP) transaction is the status phase. If `pUrb->UrbControlIn.nBuffer` is greater than zero, the sequence of transactions shall be SETUP / IN+ / OUT(ZLP), where the OUT(ZLP) transaction is the status phase.

If this request is targeting the function's default pipe, then the `wLength` field in `pUrb->UrbControlIn` must be equal to `pUrb->UrbControlIn.nBuffer`.

The number of bytes actually read is returned in `pUrb->UrbConrolIn.nActual`.

Table 13. Additional fields for USBPUMP_URB_CONTROL_IN

Name	Description
`USBPUMP_USBDI_PIPE_HANDLE hPipe;`	Specifies the target pipe – this is obtained from the result of `USBPUMP_URB_RQ_DEFINE_CONFIG`, or is the default pipe handle obtained during initialization.
`UINT32 TransferFlags;`	The transfer flags. See Table 14 for the applicable flags
`VOID *pBuffer;`	Pointer to base of user buffer.
`BYTES nBuffer;`	Number of bytes in the buffer.
`BYTES nActual;`	Output: the number of bytes actually transferred to the buffer.
`USETUP_WIRE Setup;`	The setup packet in the format it will be transferred to the target pipe.

Table 14. Transfer flags for Control In transfers

Name	Description
USBPUMP_URB_TRANSFER_FLAG_SHORT_OK	If set, a short packet (with size less than wMaxPacketSize for this pipe) is acceptable during the data-IN phase (although it will cause the transfer to be retired). If clear, a short packet will cause the transfer to be retired with an error.
USBPUMP_URB_TRANSFER_FLAG_LINKED	If set, the URB is the first of a set of linked URBs. pUrb->UrbHdr.pNext points to the head, and pUrb->UrbHdr.pLast points to the tail of the circular queue of URBs.

6.9 USBPUMP_URB_RQ_CONTROL_OUT

This request causes USBDI to issue a control-OUT transfer to a specified control pipe (possibly the default pipe). The SETUP packet is transported within the URB. If the request targets the default pipe, and is a standard request, additional processing is done as needed to ensure that the system-wide invariants with respect to the USB bus are preserved. If the function issuing a request is part of a composite device, then the request may be emulated or blocked as necessary.

This request takes a USBPUMP_URB_CONTROL_OUT structure as its parameter. The additional fields are shown in Table 15.

According to the control transfer protocol outlined in section 8.5.3 of [USBCORE], if pUrb->UrbControlOut.nBuffer is zero, then the sequence of transactions on the bus shall be SETUP / IN(ZLP), where the IN(ZLP) transaction is the status phase. If pUrb->UrbControlOut.nBuffer is greater than zero, the sequence of transactions shall be SETUP /OUT+ / IN(ZLP), where the IN(ZLP) transaction is the status phase.

There is no way to force USBDI to send a ZLP during the OUT phase.

If this request is targeting the function's default pipe, then the wLength field in pUrb->UrbControlOut must be equal to pUrb->UrbControlOut.nBuffer.

The number of bytes actually read is returned in pUrb->UrbConrolOut.nActual.

Table 15. Additional fields for USBPUMP_URB_CONTROL_OUT

Name	Description
USBPUMP_USBDI_PIPE_HANDLE hPipe;	Specifies the target pipe – this is obtained from the result of USBPUMP_URB_RQ_DEFINE_CONFIG, or is the default pipe handle obtained during initialization.

Name	Description
UINT32 TransferFlags;	The transfer flags. See Table 16 for the applicable flags
VOID *pBuffer;	Pointer to base of user buffer.
BYTES nBuffer;	Number of bytes in the buffer.
BYTES nActual;	Output: the number of bytes actually transferred from the buffer.
USETUP_WIRE Setup;	The setup packet in the format it will be transferred to the target pipe.

Table 16. Transfer flags for Control Out transfers

Name	Description
USBPUMP_URB_TRANSFER_FLAG_SHORT_OK	If set, a short packet (with size less than wMaxPacketSize for this pipe) is acceptable during the data-IN phase (although it will cause the transfer to be retired). If clear, a short packet will cause the transfer to be retired with an error.
USBPUMP_URB_TRANSFER_FLAG_LINKED	If set, the URB is the first of a set of linked URBs. pUrb->UrbHdr.pNext points to the head, and pUrb->UrbHdr.pLast points to the tail of the circular queue of URBs.

6.10 USBPUMP_URB_RQ_ISOCH_IN

This request causes USBDI to read data from a specified isochronous pipe.

This request takes a USBPUMP_URB_ISOCH_IN structure as its parameter. The additional fields are shown in Table 17.

Table 17. Additional fields for USBPUMP_URB_ISOCH_IN

Name	Description
USBPUMP_USBDI_PIPE_HANDLE hPipe;	Specifies the target pipe – this is obtained from the result of USBPUMP_URB_RQ_DEFINE_CONFIG.
UINT32 TransferFlags;	The transfer flags. See Table 18 for the applicable flags
VOID *pBuffer;	Pointer to base of user buffer.
BYTES nBuffer;	Number of bytes in the buffer.

Name	Description
`BYTES nActual;`	Output: the number of bytes actually transferred to the buffer.
`USBPUMP_ISOCH_PACKET_DESCR *pIsochDescr;`	Pointer to buffer containing the packet-by-packet isochronous descriptor information. See 6.3.1 and Table 7 for a description of this structure and its use.
`BYTES IsochDescrSize;`	Size of the buffer containing the isochronous descriptors.
`UINT32 IsochStartFrame;`	Input: The frame to use as the starting frame for the transfer. Output: the actual starting frame number
`BYTES nIsochErrs;`	Number of Isoch errors.

Table 18. Transfer flags for Isochronous Reads

Name	Description
`USBPUMP_URB_TRANSFER_FLAG_SHORT_OK`	If set, a short packet (with size less than `wMaxPacketSize` for this pipe) is acceptable, and will not retire the transfer. If clear, a short packet will cause the transfer to be retired with an error.
`USBPUMP_URB_TRANSFER_FLAG_ASAP`	If set, the transfer in this URB shall be started as soon as possible. Otherwise, the starting frame number is taken from `pUrb->UrbIsochIn.IsochStartFrame`.

6.11 USBPUMP_URB_RQ_ISOCH_OUT

This request causes USBDI to write data to a specified isochronous pipe.

This request takes a USBPUMP_URB_ISOCH_OUT structure as its parameter. The additional fields are shown in Table 19.

Table 19. Additional fields for USBPUMP_URB_ISOCH_OUT

Name	Description
`USBPUMP_USBDI_PIPE_HANDLE hPipe;`	Specifies the target pipe – this is obtained from the result of USBPUMP_URB_RQ_DEFINE_CONFIG.
`UINT32 TransferFlags;`	The transfer flags. See Table 20 for the applicable flags
`VOID *pBuffer;`	Pointer to base of transfer buffer.
`BYTES nBuffer;`	Number of bytes in the buffer.

Name	Description
`BYTES nActual;`	Output: the number of bytes actually transferred to the buffer.
`USBPUMP_ISOCH_PACKET_DESCR *pIsochDescr;`	Pointer to buffer containing the packet-by-packet isochronous descriptor information. See 6.3.1 and Table 7 for a description of this structure and its use.
`BYTES IsochDescrSize;`	Size of the buffer containing the isochronous descriptors.
`UINT32 IsochStartFrame;`	Input: The frame to use as the starting frame for the transfer. Output: the actual starting frame number
`BYTES nIsochErrs;`	Number of Isoch errors.

Table 20. Transfer flags for Isochronous Writes

Name	Description
`USBPUMP_URB_TRANSFER_FLAG_ASAP`	If set, the transfer in this URB shall be started as soon as possible. Otherwise, the starting frame number is taken from `pUrb->UrbIsochIn.IsochStartFrame`.

6.12 USBPUMP_URB_RQ_DEFINE_CONFIG

This request asks USBD and the associated HCD to allocate sufficient resources to operate the USB device in any of the described configurations. The request takes a `USBPUMP_URB_DEFINE_CONFIG` structure as its argument; this structure in turn points to a buffer containing a device configuration definition tree. The fields of the structure appear in Table 21.

Be advised that the memory used by `USBPUMP_URB_DEFINE_CONFIG` request scales with the complexity of the device description. Client drivers using this routine are advised to examine the structure and limit the scope of the actual `USBPUMP_URB_DEFINE_CONFIG` structure to the pipes that the client driver intends to use.

When class drivers are created using `UsbPumpUsbdiLib_CreateDriverClass()` (section 3.3.1), USBDI pre-allocates USBDI pipe-management objects based on the class driver's declared "maximum number of pipes per instance". This means that as long as the class driver restricts itself to that maximum number of pipe slots in the configuration tree, USBDI will be able to use those pre-allocated pipes to satisfy the request.

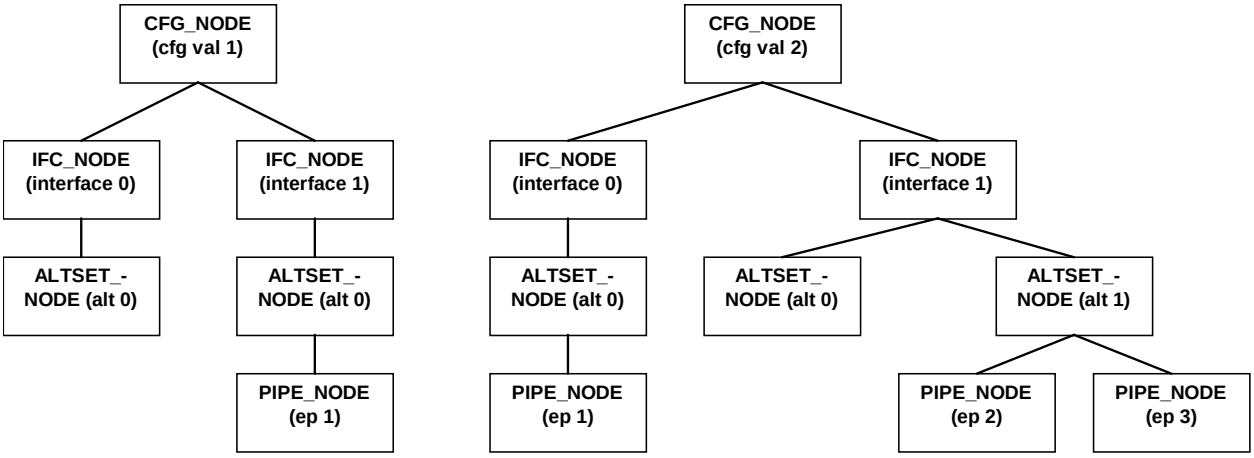
Notice, please, please that this maximum number of pipes includes all pipes that are defined in alternate settings. This is true even though the pipes are not

> in immediate use; they're needed for bookkeeping purposes, and to ensure that a pipe will be available after SET_INTERFACE or SET_CONFIG.

The configuration tree is prepared by the client, and is encoded in preorder sequential representation ([Knuth1973] 2.3.3). The tree consists of heterogeneous nodes in a well-defined hierarchy.

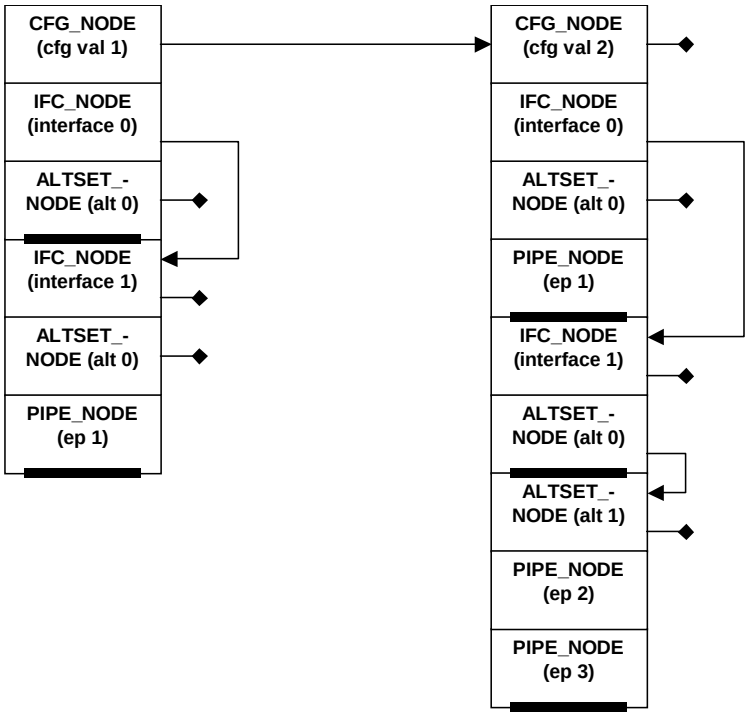
At the top of the tree, we find one or more `USBPUMP_USBDI_CFG_NODE`s. These serve to represent the possible configuration settings. Each `USBPUMP_USBDI_CFG_NODE` is linked to its sibling via a relative link (`iNextCfgNode`), and is followed immediately (sequentially in memory) by its first child (if such exists), which is always a `USBPUMP_USBDI_IFC_NODE`. To adapt a figure from [Knuth1973], Figure 4 shows a sample device tree for a device with two configurations, and Figure 5 shows how the tree is represented to USBDI. The first configuration has two interfaces, with no endpoints in the first interface, and with one endpoint in the second. The second configuration has one endpoint in the first interface, and two alternate settings in the second interface, one of which has two endpoints.

Figure 4. Sample Device Tree



For convenience, in Figure 5 we have shown the nodes in two columns; however, in practice they appear in ascending locations in memory. The heavy bars indicate a break in pre-order traversal of the tree; the arrows indicate link fields in the data structures. Note that `CFG_NODE`s, `IFC_NODE`s, and `ALTSET_NODE`s all have link fields.

Figure 5. Configuration as Represented in Memory



Also, note that `ALTSET_NODE`s exist even in the case where the interface has no alternate settings, apart from alternate setting zero, and even when there is no endpoint associated with the alternate setting.

Macros are defined to assist in traversing these data structures. See the header file `"usbpump_usbdi_request.h"` for details.

To release the memory used by a defined configuration, client drivers may send a `USBPUMP_URB_RQ_DEFINE_CONFIG` with `UrbDefineConfig.pRootConfigNode == NULL` or with `UrbDefineConfig.sizeConfigBuffer == 0`. This will release all resources and prevent further transfers on non-default pipes. The device will remain in its currently configured state; to return the device to the Addressed state, clients may send `SET_CONFIGURATION(0)`.

Table 21. Additional fields for USBPUMP_URB_DEFINE_CONFIG

Name	Description
`USBPUMP_USBDI_CFG_NODE *pRootConfigNode;`	Pointer to the base of a buffer containing a tree that describes the possible operating configurations of the device.
`BYTES sizeConfigBuffer;`	The size of the configuration buffer, expressed in bytes. The entire configuration tree must fit in the buffer that starts at `*pRootConfigNode` and runs for `sizeConfigBuffer` bytes.

Table 22. USBPUMP_USBDI_CFG_NODE

Name	Description
`UINT16 iNextCfgNode;`	Byte index from the base of this node to the next configuration node. Zero indicates that this is the last configuration.
`UINT8 nInterfaces;`	Number of interfaces defined within this configuration. This is a convenience, as this can also be learned by traversing the data structures.
`UINT8 bConfigurationValue;`	The configuration value that selects this configuration
`UINT8 bmAttributes;`	The `bmAttributes` field as defined for configuration descriptors by [USBCORE]. Bit 6, if set, indicates that the device is self-powered in this configuration. Bit 5, if set, indicates that the device supports remote wakeup in this configuration.
`UINT8 bMaxPower;`	The maximum power needed in this configuration, expressed in units of 2mA.

Table 23. USBPUMP_USBDI_IFC_NODE

Name	Description
`UINT16 iNextIfcNode;`	Byte index from the base of this node to the next sibling interface node. If zero, this is the last sibling in this configuration.
`UINT16 iCurrentAltSetting;`	
`UINT8 bInterfaceNumber;`	Interface number. These will always appear in ascending order, but might not be sequential. (Some devices do not have sequential interface numbers, and composite devices will see filtered descriptors that omit interfaces that belong to other functions.
`UINT8 bNumAltSettings;`	Number of alternate settings, for convenience. Must be at least 1.

Table 24. USBPUMP_USBDI_ALTSET_NODE

Name	Description
`UINT16 iNextAltSetNode;`	Byte index from the base of this node to the next sibling alternate setting node. If zero, this is the last sibling in this

Name	Description
	interface.
UINT8 bAlternateSetting;	The alternate setting. This will always appear in ascending order.
UINT8 bNumPipes;	The number of pipes in this alternate setting. This is **not** just for convenience: this is how traversal software learns how many PIPE_NODEs there actually are. The traversal software does not assume that the data structure is packed.

Table 25. USBPUMP_USBDI_PIPE_NODE

Node	Description
UINT8 bEndpointAddress;	The target endpoint address (in bits 3..0) and direction (in bit 7). Other bits are reserved.
UINT8 bmAttributes;	The endpoint type (in bits 1..0). Other bits are reserved. Note that other bits from the high-speed endpoint descriptor bmAttributes field are not used in this field.
UINT8 bExtraPackets;	For high-speed interrupt and isochronous pipes, this field represents the additional transactions per frame. The extra transactions are taken from wMaxPacketSize bits 12..11.
UINT16 wInterval more q;	For periodic endpoints, this specifies the polling interval for the endpoint in frames or microframes. Unlike the value in the endpoint descriptor, this value is directly expressed in terms of the polling interval, and therefore ranges from 0 to 32768 (representing an Isochronous wInterval value of 16). The library routine UsbPumpLib_Calculate-NormalizedInterval() can be used to assist clients in generating this value from an endpoint descriptor.
UINT16 wMaxPacketSize;	This represents the actual max packet size on the bus, i.e. bits 10..0 of the wMaxPacketSize in the endpoint descriptor. The bits from bExtraTransactions are not stored here.
UINT32 dwMaxTransferSize;	Maximum transfer size for this pipe. USBD and the HCD use this to allocate system DMA control resources needed to support a transfer of this size.
USBPUMP_USBDI_PIPE_HANDLE hPipe;	Output: the pipe handle that can be used to refer to this pipe once the pipe has been activated by a SET_CONFIGURATION or SET_INTERFACE command. If the pipe is not active, requests sent to the pipe will be

Node	Description
	rejected.

6.12.1 Bandwidth Allocation Rules

MCCI's USB stack uses a different policy for bandwidth allocation than most USB stacks. Most stacks do bandwidth allocation based on the specific configuration or interface selected. However, the DataPump USBDI pre-allocates bandwidth for periodic transfers based on the worst-case possibility when the configuration is defined by `DEFINE_CONFIG`. This means that the user is immediately aware (when the device is plugged in) whether the device will operate in all its modes; and the driver writer does not need to worry about `SET_INTERFACE` or `SET_CONFIG` failing.

For example, if the client submits a `DEFINE_CONFIG` with an interface with two alternate settings, USBDI will reserve enough bandwidth for the client to allow either alternate setting to be used. If the client submits a `DEFINE_CONFIG` with two interfaces, each with two alternate settings, USBDI will reserve enough bandwidth to allow both interfaces to be used concurrently in either alternate setting.

In rare circumstances, this may cause unexpected effects. For example, if a full-speed device has two interfaces, each with an alternate setting having an isochronous endpoint with 1023 byte `wMaxPacketSize`, then it will not be possible to successfully do a `DEFINE_CONFIG` unless the client deletes one of the alternate settings from the input.

`SUGGEST_CONFIG` doesn't take into account bandwidth availability; it merely performs a parsing function based on the descriptors of the device. Thus, `SUGGEST_CONFIG` may suggest a configuration that (in fact) cannot be used. Class drivers that need to work with this kind of device will have to filter the configuration descriptors themselves.

(Similarly, `SUGGEST_CONFIG` doesn't take into account the amount of power available at the port for the device.)

6.13 USBPUMP_URB_RQ_SUGGEST_CONFIG

This request asks USBD to build a suggested configuration structure in the users buffer, based on the descriptors of the device. The suggested configuration can then be used in a subsequent `DEFINE_CONFIG` request. This request is provided primarily as a convenience for class drivers that are proxying for remote clients, who otherwise might have to replicate all the descriptor parsing logic that is already available to local clients of USBDI through the DataPump libraries. However, local class drivers may also use this request in order to simplify their implementation.

This request takes a buffer as its input, and prepares a `USBPUMP_URB_DEFINE_CONFIG` structure as its result. The request-specific fields are shown in Table 26.

The actual length of the suggested configuration is returned to the caller in `pUrb->UrbDefineConfiguration.sizeConfigBuffer`. The returned pipe handles will be zero.

Table 26. Request-Specific fields for `USBPUMP_URB_SUGGEST_CONFIG`

Field	Description
`VOID *pConfigBuffer;`	Pointer to the buffer into which USBDI will build the configuration tree structure.
`BYTES sizeConfigBuffer;`	Maximum number of bytes to be written into the configuration buffer.
`BYTES nActualConfigBuffer;`	Output: set to the number of bytes written to the configuration buffer.
`CONST VOID *pOptionalConfigBundles;`	Optional pointer to configuration bundles. This may be used to override `USBPUMP_URB_SUGGEST_CONFIG`'s normal behavior of reading configuration descriptors from the device.
`BYTES sizeConfigBundles;`	The size of the configuration bundles.

The normal configuration flow when using this request is:

1. USBDI assigns an instance of a device to a class driver, and notifies the class driver.

2. The class driver has a pre-allocated, empty buffer for building the configuration tree. It passes a pointer to that buffer, and its size, to USBDI using `USBPUMP_URB_RQ_SUGGEST_CONFIG`.

3. If `pOptionalConfigBundles` is `NULL`, USBDI reads the configuration descriptors from the device, and uses the device's configuration bundles to fill in the configuration information in the class driver's buffer. A single function of a multi-function device will only see one configuration; a device-level driver will get back information for all the configurations available in the device.

   On the other hand, if `pOptionalConfigBundles` is not `NULL`, then USBDI uses `pOptionalConfigBundles` as a pointer to a buffer containing the configuration bundles for this device. Each configuration bundle must be packed sequentially into the supplied buffer. This option has two uses. First, it can be used to provide substitute descriptors for broken devices. Second, it can be used in cases where the standard USBDI configuration descriptor buffer is not large enough to handle the descriptors of the device.

4. The class driver uses the information returned by `USBPUMP_URB_RQ_SUGGEST_CONFIG` to create and submit a `USBPUMP_URB_RQ_DEFINE_CONFIG` request (see section 6.12).

5. USBDI allocates resources based on the submitted information, if possible. These resources include pipe handles, and tracking information for possible configurations and alternate settings. The tree passed down in `USBPUMP_URB_RQ_DEFINE_CONFIG` is updated to include the assigned pipe handles. These pipe handles are valid, but not yet ready for use.

6. The class driver then issues a `USBPUMP_URB_RQ_CONTROL_OUT` request carrying a `SET_CONFIG` command to activate one of the configurations defined by the previous `USBPUMP_URB_UR_DEFINE_CONFIG` request. This activates all the pipes in alternate settings zero of all the interfaces defined in the selected configuration. USBDI will reject an attempt to set an undefined configuration.

7. The class driver, if needed, issues `USBPUMP_URB_RQ_CONTROL_OUT` requests carrying `SET_INTERFACE` commands to activate any non-zero alternate settings.

8. The class driver uses activated pipe handles (returned in step 5) to perform its operations.

USBDI does not check (at step 4) whether the configuration information matches the devices descriptors. This allows the class driver to reduce the amount of resources required within USBDI by eliminating unnecessary pipes, interfaces, and configurations before submitting the request.

6.14 USBPUMP_URB_RQ_GET_DEVICE_INFO

This request asks USBD to fill in information about the resources being consumed by the device (or function, in case this is a composite device).

This request takes a `USBPUMP_URB_GET_DEVICE_INFO` structure as its parameter. The request-specific fields are shown in Table 27.

Table 27. Request-Specific fields for `USBPUMP_URB_GET_DEVICE_INFO`

Field	Description
`USBPUMP_USBDI_PIPE_HANDLE hDefaultPipe;`	Output: the handle for the default pipe.
`USBPUMP_DEVICE_SPEED DeviceSpeed;`	Output: The speed of the device: either `USBPUMP_DEVICE_SPEED_LOW`, `USBPUMP_DEVICE_SPEED_FULL`, or `USBPUMP_DEVICE_SPEED_HIGH`.
`UINT32 DeviceBandwidth;`	Output: The device bandwidth consumed, expressed in bus clocks per second.
`UINT32 BusBandwidth;`	Output: The amount of bus bandwidth in use total, expressed in bus clocks per second.

Field	Description
`UINT32 BusClockHertz;`	Output: the relevant number of bus clocks per second. For some device connect technologies, this is likely to be set by the root bus connect speed. On the other hand, for native Wireless USB hosts and devices, this will be nominally 480 MBps.
`USBPUMP_USBDI_DEVICE_HANDLE hHub;`	Output: The device handle of the parent hub.
`UINT32 iPort`	Output: the port index for this device on the parent hub.
`UINT32 RequestLatency;`	Output: the estimated request latency in (micro)frames.

6.15 USBPUMP_URB_RQ_GET_FRAME

This request asks USBD to return the current frame number or microframe number for the bus on which the port resides.

This request takes a `USBPUMP_URB_GET_FRAME` structure as its parameter. The request-specific fields are shown in Table 28.

Table 28. Request-Specific fields for `USBPUMP_URB_GET_FRAME`

Field	Description
`UINT32 StandardFrame;`	Output: the "standard" frame count, in milliseconds (always compatible with full speed), as a 32-bit number.
`UINT64 NativeFrame;`	Output: the "native" frame count, (microframes for high speed, frames for full speed) -- as a 64-bit number
`UINT32 Numerator;`	Output: numerator and denominator for converting native frame count to standard frame count: 1/1 for full speed, 1/8 for high speed, and other values for Wireless USB or interchip USB. (Native Wireless USB is 1000/1024, simplified to 125/128; and for interchip USB there are a number of possible values.)
`UINT32 Denominator;`	Output: See above.

6.16 USBPUMP_URB_RQ_GET_PORT_STATUS

This request asks USBD to return the current status of the port.

This request takes a `USBPUMP_URB_GET_PORT_STATUS` structure as its parameter. The request-specific fields are shown in Table 29.

Table 29. Request-Specific fields for `USBPUMP_URB_GET_PORT_STATUS`

Field	Description
`UINT32 PortStatus;`	Output: the current port status. See Table 30 for the bits in this field.

Table 30. Port Status bits

Name	Description
`USB_Hub_PORT_STATUS_wStatus_Connect`	If set, then the port is connected.
`USB_Hub_PORT_STATUS_wStatus_Enable`	If set, then the port is enabled. It might be disabled due to babble detect.
`USB_Hub_PORT_STATUS_wStatus_Suspend`	If set, then the port has been suspended.
`USB_Hub_PORT_STATUS_wStatus_OverCurrent`	If set, then the port has experienced an over-current trip.
`USB_Hub_PORT_STATUS_wStatus_Reset`	If set, the port has been reset.
`USB_Hub_PORT_STATUS_wStatus_Power`	If set, then port power is on.
`USB_Hub_PORT_STATUS_wStatus_LowSpeed`	If set, the port is in Low Speed mode.
`USB_Hub_PORT_STATUS_wStatus_HighSpeed`	If set, the port is in High Speed mode.
`USB_Hub_PORT_STATUS_wStatus_Test`	If set, then the port is in test mode.
`USB_Hub_PORT_STATUS_wStatus_Manual`	If set, then the indicator for this port is in manual mode.

6.17 USBPUMP_URB_RQ_REENUMERATE_PORT

This request asks USBD to reset and re-enumerate the port.

This request takes a `USBPUMP_URB_DEVICE_CONTROL` structure as its parameter. There are no request-specific parameters.

The request completes as soon as it has been communicated to the appropriate agent in the system. The client driver will receive a device departure notification, and the instance will be returned to the class driver pool.

6.18 USBPUMP_URB_RQ_RESET_PIPE

This request causes USBDI to reset the specified pipe. Any halt status in the pipe is reset.

This request takes a `USBPUMP_URB_PIPE_CONTROL` as its parameter. The request-specific fields are shown in Table 8.

Unlike some implementations, this request only resets the endpoint-halted flag in the pipe database. It doesn't reset the data toggles. Therefore, this request is rarely used. To reset the data toggle and clear the endpoint-halted flag, use `USBPUMP_URB_RQ_CONTROL_OUT` to send `CLEAR_FEATURE(ENDPOINT_HALT)`.

Unlike some other implementations, this request should not be used on the default pipe. The endpoint-halted flag will never be set on the default pipe. If the device gets into a persistent stall state on endpoint 0, send `CLEAR_FEATURE(ENDPOINT_HALT)` to the default pipe.

6.19 USBPUMP_URB_RQ_RESET_PORT

This request asks USBD to reset any pending error conditions and re-enable the port.

This request takes a `USBPUMP_URB_DEVICE_CONTROL` structure as its parameter. There are no request-specific parameters.

USBDI sends a reset to the device in question and performs a set address and the appropriate set configuration and set interface. Any pending requests to the device are retired with "not responding status".

7 URB Preparation Functions

7.1 UsbPumpUrb_PrepareAbortPipe()

```
VOID UsbPumpUrb_PrepareAbortPipe(
          USBPUMP_URB *pUrb,
          USBPUMP_URB_LENGTH UrbSize,
          USBPUMP_USBDI_PORT_KEY PortKey,
          USBPUMP_USBDI_TIMEOUT HcdTimeout,
          USBPUMP_USBDI_PIPE_HANDLE hPipe
          );
```

Abort all pending transfers for the pipe specified by hPipe.

7.2 UsbPumpUrb_PrepareBulkIntIn()

```
VOID UsbPumpUrb_PrepareBulkIntIn(
          USBPUMP_URB *pUrb,
          USBPUMP_URB_LENGTH UrbSize,
          USBPUMP_USBDI_PORT_KEY PortKey,
          USBPUMP_USBDI_TIMEOUT HcdTimeout,
          USBPUMP_USBDI_PIPE_HANDLE hPipe,
          VOID *pBuffer,
```

```
          BYTES nBuffer,
          ARG_UINT32 TransferFlags
          );

VOID UsbPumpUrb_PrepareBulkIntIn_V2(
          USBPUMP_URB *pUrb,
          USBPUMP_URB_LENGTH UrbSize,
          USBPUMP_USBDI_PORT_KEY PortKey,
          USBPUMP_USBDI_TIMEOUT HcdTimeout,
          USBPUMP_USBDI_PIPE_HANDLE hPipe,
          VOID *pBuffer,
          BYTES nBuffer,
          USBPUMP_BUFFER_HANDLE hBuffer,
          ARG_UINT32 TransferFlags
          );
```

Prepare a USBPUMP_URB_RQ_BULKINT_IN request, used reading from a bulk or interrupt pipe.

7.3 UsbPumpUrb_PrepareBulkIntOut()

```
VOID UsbPumpUrb_PrepareBulkIntOut(
          USBPUMP_URB *pUrb,
          USBPUMP_URB_LENGTH UrbSize,
          USBPUMP_USBDI_PORT_KEY PortKey,
          USBPUMP_USBDI_TIMEOUT HcdTimeout,
          USBPUMP_USBDI_PIPE_HANDLE hPipe,
          CONST VOID *pBuffer,
          BYTES nBuffer,
          ARG_UINT32 TransferFlags
          );

VOID UsbPumpUrb_PrepareBulkIntOut_V2(
          USBPUMP_URB *pUrb,
          USBPUMP_URB_LENGTH UrbSize,
          USBPUMP_USBDI_PORT_KEY PortKey,
          USBPUMP_USBDI_TIMEOUT HcdTimeout,
          USBPUMP_USBDI_PIPE_HANDLE hPipe,
          CONST VOID *pBuffer,
          BYTES nBuffer,
          USBPUMP_BUFFER_HANDLE hBuffer,
          ARG_UINT32 TransferFlags
          );
```

Prepare a USBPUMP_URB_RQ_BULKINT_OUT request, used writing to a bulk or interrupt pipe.

7.4 UsbPumpUrb_PrepareControlIn()

```
VOID UsbPumpUrb_PrepareControlIn(
          USBPUMP_URB *pUrb,
          USBPUMP_URB_LENGTH UrbSize,
          USBPUMP_USBDI_PORT_KEY PortKey,
          USBPUMP_USBDI_TIMEOUT HcdTimeout,
          USBPUMP_USBDI_PIPE_HANDLE hPipe,
          VOID *pBuffer,
          BYTES nBuffer,
          ARG_UINT32 TransferFlags,
          CONST USETUP_WIRE *pSetup
          );

VOID UsbPumpUrb_PrepareControlIn_V2(
          USBPUMP_URB *pUrb,
          USBPUMP_URB_LENGTH UrbSize,
          USBPUMP_USBDI_PORT_KEY PortKey,
          USBPUMP_USBDI_TIMEOUT HcdTimeout,
          USBPUMP_USBDI_PIPE_HANDLE hPipe,
          VOID *pBuffer,
          BYTES nBuffer,
          USBPUMP_BUFFER_HANDLE hBuffer,
          ARG_UINT32 TransferFlags,
          CONST USETUP_WIRE *pSetup
          );
```

Prepare a USBPUMP_URB_RQ_CONTROL_IN request, used reading from a control pipe, especially the default pipe.

7.5 UsbPumpUrb_PrepareControlOut()

```
VOID UsbPumpUrb_PrepareControlOut(
          USBPUMP_URB *pUrb,
          USBPUMP_URB_LENGTH UrbSize,
          USBPUMP_USBDI_PORT_KEY PortKey,
          USBPUMP_USBDI_TIMEOUT HcdTimeout,
          USBPUMP_USBDI_PIPE_HANDLE hPipe,
          CONST VOID *pBuffer,
          BYTES nBuffer,
          ARG_UINT32 TransferFlags,
          CONST USETUP_WIRE *pSetup
          );

VOID UsbPumpUrb_PrepareControlOut_V2(
          USBPUMP_URB *pUrb,
          USBPUMP_URB_LENGTH UrbSize,
          USBPUMP_USBDI_PORT_KEY PortKey,
          USBPUMP_USBDI_TIMEOUT HcdTimeout,
```

```
        USBPUMP_USBDI_PIPE_HANDLE hPipe,
        CONST VOID *pBuffer,
        BYTES nBuffer,
        USBPUMP_BUFFER_HANDLE hBuffer,
        ARG_UINT32 TransferFlags,
        CONST USETUP_WIRE *pSetup
        );
```

Prepare a USBPUMP_URB_RQ_CONTROL_OUT request, used writing to a bulk or interrupt pipe.

7.6 UsbPumpUrb_PrepareDefineConfig()

```
VOID UsbPumpUrb_PrepareDefineConfig(
        USBPUMP_URB *pUrb,
        USBPUMP_URB_LENGTH UrbSize,
        USBPUMP_USBDI_PORT_KEY PortKey,
        USBPUMP_USBDI_TIMEOUT HcdTimeout,
        USBPUMP_USBDI_CFG_NODE *pRootConfigNode,
        BYTES sizeConfigBuffer
        );
```

Prepare a USBPUMP_URB_RQ_DEFINE_CONFIG request, this is used to allocate bandwidth for a device on a bus, and power from a hub.

7.7 UsbPumpUrb_PrepareGetDeviceInfo()

```
VOID UsbPumpUrb_PrepareGetDeviceInfo(
        USBPUMP_URB *pUrb,
        USBPUMP_URB_LENGTH UrbSize,
        USBPUMP_USBDI_PORT_KEY PortKey,
        USBPUMP_USBDI_TIMEOUT HcdTimeout
        );
```

Prepare a USBPUMP_URB_RQ_GET_DEVICE_INFO request, which is used to get information about the device from USBDI.

7.8 UsbPumpUrb_PrepareGetFrame()

```
VOID UsbPumpUrb_PrepareGetFrame(
        USBPUMP_URB *pUrb,
        USBPUMP_URB_LENGTH UrbSize,
        USBPUMP_USBDI_PORT_KEY PortKey,
        USBPUMP_USBDI_TIMEOUT HcdTimeout
        );
```

Prepare a USBPUMP_URB_RQ_GET_FRAME request, which is used to get the current frame number (well, really the frame number sometime in the past, by the time the URB completes).

7.9 UsbPumpUrb_PrepareGetPortStatus()

```
VOID UsbPumpUrb_PrepareGetPortStatus(
          USBPUMP_URB *pUrb,
          USBPUMP_URB_LENGTH UrbSize,
          USBPUMP_USBDI_PORT_KEY PortKey,
          USBPUMP_USBDI_TIMEOUT HcdTimeout
          );
```

Prepare a USBPUMP_URB_RQ_GET_PORT_STATUS request, which is used to get the current port status.

7.10 UsbPumpUrb_PrepareIsochIn()

```
VOID UsbPumpUrb_PrepareIsochIn(
          USBPUMP_URB *pUrb,
          USBPUMP_URB_LENGTH UrbSize,
          USBPUMP_USBDI_PORT_KEY PortKey,
          USBPUMP_USBDI_TIMEOUT HcdTimeout,
          USBPUMP_USBDI_PIPE_HANDLE hPipe,
          VOID *pBuffer,
          BYTES nBuffer,
          ARG_UINT32 TransferFlags,
          USBPUMP_ISOCH_PACKET_DESCR *pIsochDescr,
          BYTES IsocDescrSize,
          UINT32 IsochStartFrame
          );

VOID UsbPumpUrb_PrepareIsochIn_V2(
          USBPUMP_URB *pUrb,
          USBPUMP_URB_LENGTH UrbSize,
          USBPUMP_USBDI_PORT_KEY PortKey,
          USBPUMP_USBDI_TIMEOUT HcdTimeout,
          USBPUMP_USBDI_PIPE_HANDLE hPipe,
          VOID *pBuffer,
          BYTES nBuffer,
          USBPUMP_BUFFER_HANDLE hBuffer,
          ARG_UINT32 TransferFlags,
          USBPUMP_ISOCH_PACKET_DESCR *pIsochDescr,
          BYTES IsocDescrSize,
          USBPUMP_BUFFER_HANDLE hIsochDescr,
          UINT32 IsochStartFrame
          );
```

Prepare a USBPUMP_URB_RQ_ISOCH_IN request, used reading from an isochronous pipe.

7.11 UsbPumpUrb_PrepareIsochOut()

```
VOID UsbPumpUrb_PrepareIsochOut(
          USBPUMP_URB *pUrb,
          USBPUMP_URB_LENGTH UrbSize,
          USBPUMP_USBDI_PORT_KEY PortKey,
          USBPUMP_USBDI_TIMEOUT HcdTimeout,
          USBPUMP_USBDI_PIPE_HANDLE hPipe,
          CONST VOID *pBuffer,
          BYTES nBuffer,
          ARG_UINT32 TransferFlags,
          USBPUMP_ISOCH_PACKET_DESCR *pIsochDescr,
          BYTES IsocDescrSize,
          UINT32 IsochStartFrame
          );

VOID UsbPumpUrb_PrepareIsochOut_V2(
          USBPUMP_URB *pUrb,
          USBPUMP_URB_LENGTH UrbSize,
          USBPUMP_USBDI_PORT_KEY PortKey,
          USBPUMP_USBDI_TIMEOUT HcdTimeout,
          USBPUMP_USBDI_PIPE_HANDLE hPipe,
          CONST VOID *pBuffer,
          BYTES nBuffer,
          USBPUMP_BUFFER_HANDLE hBuffer,
          ARG_UINT32 TransferFlags,
          USBPUMP_ISOCH_PACKET_DESCR *pIsochDescr,
          BYTES IsocDescrSize,
          USBPUMP_BUFFER_HANDLE hIsochDescr,
          UINT32 IsochStartFrame
          );
```

Prepare a USBPUMP_URB_RQ_ISOCH_OUT request, used writing from an isochronous pipe.

7.12 UsbPumpUrb_PrepareReenumeratePort()

```
VOID UsbPumpUrb_PrepareReenumeratePort(
          USBPUMP_URB *pUrb,
          USBPUMP_URB_LENGTH UrbSize,
          USBPUMP_USBDI_PORT_KEY PortKey,
          USBPUMP_USBDI_TIMEOUT HcdTimeout
          );
```

Prepare a USBPUMP_URB_RQ_REENUMERATE_PORT request, which is used to force USBDI to reenumerate a port.

7.13 UsbPumpUrb_PrepareResetPipe()

```
VOID UsbPumpUrb_PrepareResetPipe(
          USBPUMP_URB *pUrb,
          USBPUMP_URB_LENGTH UrbSize,
          USBPUMP_USBDI_PORT_KEY PortKey,
          USBPUMP_USBDI_TIMEOUT HcdTimeout,
          USBPUMP_USBDI_PIPE_HANDLE hPipe
          );

VOID UsbPumpUrb_PrepareResetPipe_V2(
          USBPUMP_URB *pUrb,
          USBPUMP_URB_LENGTH UrbSize,
          USBPUMP_USBDI_PORT_KEY PortKey,
          USBPUMP_USBDI_TIMEOUT HcdTimeout,
          USBPUMP_USBDI_PIPE_HANDLE hPipe,
          UINT32 ResetPipeFlags
          );
```

Prepare a USBPUMP_URB_RQ_RESET_PIPE request, which is used to reset the data toggles and clear any halt condition for the pipe specified by hPipe.

7.14 UsbPumpUrb_PrepareResetPort()

```
VOID UsbPumpUrb_PrepareResetPort(
          USBPUMP_URB *pUrb,
          USBPUMP_URB_LENGTH UrbSize,
          USBPUMP_USBDI_PORT_KEY PortKey,
          USBPUMP_USBDI_TIMEOUT HcdTimeout
          );
```

Prepare a USBPUMP_URB_RQ_RESET_PORT request, which is used to clear any error conditions (babble detect, etc.) that apply to the port.

7.15 UsbPumpUrb_PrepareSuggestConfig()

```
VOID UsbPumpUrb_PrepareSuggestConfig(
          USBPUMP_URB *pUrb,
          USBPUMP_URB_LENGTH UrbSize,
          USBPUMP_USBDI_PORT_KEY PortKey,
          USBPUMP_USBDI_TIMEOUT HcdTimeout,
          CONST VOID *pRootConfigNode,
          BYTES sizeConfigBuffer,
          VOID *pOptionalConfigBundles,
          BYTES sizeConfigBundles
          );
```

Prepare a USBPUMP_URB_RQ_SUGGEST_CONFIG request. See 6.13, above.

7.16 UsbPumpUrb_PrepareBulkIntStreamIn()

```
VOID UsbPumpUrb_PrepareBulkIntStreamIn(
          USBPUMP_URB *pUrb,
          USBPUMP_URB_LENGTH UrbSize,
          USBPUMP_USBDI_PORT_KEY PortKey,
          USBPUMP_USBDI_TIMEOUT HcdTimeout,
          USBPUMP_USBDI_PIPE_HANDLE hPipe,
          CONST VOID *pBuffer,
          BYTES nBuffer,
          USBPUMP_BUFFER_HANDLE hBuffer,
          ARG_UINT32 TransferFlags,
          UINT16 StreamID
          );
```

Prepare a USBPUMP_URB_RQ_BULKINT_STREAM_IN request.

7.17 UsbPumpUrb_PrepareBulkIntStreamOut()

```
VOID UsbPumpUrb_PrepareBulkIntStreamOut(
          USBPUMP_URB *pUrb,
          USBPUMP_URB_LENGTH UrbSize,
          USBPUMP_USBDI_PORT_KEY PortKey,
          USBPUMP_USBDI_TIMEOUT HcdTimeout,
          USBPUMP_USBDI_PIPE_HANDLE hPipe,
          CONST VOID *pBuffer,
          BYTES nBuffer,
          USBPUMP_BUFFER_HANDLE hBuffer,
          ARG_UINT32 TransferFlags,
          UINT16 StreamID
          );
```

Prepare a USBPUMP_URB_RQ_BULKINT_STREAM_OUT request.

7.18 Generic URB Preparation Routines

The routines in this section may be used to prepare URBs of a specified kind, while filling in the URB request code. These are documented for completeness, but are not intended for use by non-MCCI code.

7.18.1 UsbPumpUrb_PrepareDeviceControl()

```
VOID UsbPumpUrb_PrepareDeviceControl(
          USBPUMP_URB *pUrb,
          USBPUMP_URB_LENGTH UrbSize,
          USBPUMP_URB_CODE RequestCode,
          USBPUMP_USBDI_PORT_KEY PortKey,
```

```
        USBPUMP_USBDI_TIMEOUT HcdTimeout
        );
```

Prepare a URB of type UrbDeviceControl, with the specified request code.

7.18.2 UsbPumpUrb_PrepareHeader()

```
VOID UsbPumpUrb_PrepareHeader(
        USBPUMP_URB *pUrb,
        USBPUMP_URB_LENGTH UrbSize,
        USBPUMP_URB_CODE RequestCode,
        USBPUMP_USBDI_PORT_KEY PortKey,
        USBPUMP_USBDI_TIMEOUT HcdTimeout
        );
```

Prepare a generic URB UrbHdr, with the specified values. Fields not specified here will be filled in by the code that processes URB submission.

7.18.3 UsbPumpUrb_PreparePipeControl()

```
VOID UsbPumpUrb_PreparePipeCode(
        USBPUMP_URB *pUrb,
        USBPUMP_URB_LENGTH UrbSize,
        USBPUMP_URB_CODE RequestCode,
        USBPUMP_USBDI_PORT_KEY PortKey,
        USBPUMP_USBDI_TIMEOUT HcdTimeout,
        USBPUMP_USBDI_PIPE_HANDLE hPipe
        );
```

Prepare a generic pipe-control URB, with the specified request code.

8 USBDI IOCTLs

A summary of the USBDI Inquiry and Control IOCTL codes is shown in Table 31. A summary of the USBDI notification IOCTL codes is shown in Table 32.

Table 31. USBDI IOCTL Inquiry and Control Codes

Name	Description
USBPUMP_IOCTL_USBDI_PORT_IDLE_FUNCTION	Sent to USBDI to notify USBDI that the function driver instance is through working with the device. This causes the driver to be detached from the device.
USBPUMP_IOCTL_USBDI_PORT_SUSPEND_FUNCTION	Sent to USBDI to notify USBDI that the function driver instance is suspending the device. To support remote wakeup, the function driver instance should issue the

Name	Description
	set-feature (RWUP) to the device prior to sending this IOCTL.
USBPUMP_IOCTL_USBDI_PORT_RESUME_FUNCTION	Sent to the USBDI to notify USBDI that the function driver instance is resuming the device.
USBPUMP_IOCTL_USBDI_PORT_SET_IDLE_OTG_PORT	Sent to the USBDI to notify USBDI that the HCD or PHY set the fSetIdle flag.
USBPUMP_IOCTL_USBDI_PORT_MATCH_AND_NOTIFY	Sent to the USBDI to notify USBDI (via a port) to match and bind a function for the port.
USBPUMP_IOCTL_USBDI_PORT_GET_PORT_STATUS	Fetches information about port status flags for a given port
USBPUMP_IOCTL_USBDI_PORT_L1_SLEEP_ASYNC	Sent to the USBDI to notify (via a port) that the function driver sleep the port.
USBPUMP_IOCTL_USBDI_PORT_SUSPEND_ASYNC	Sent to the USBDI to notify that the function driver suspend the port.
USBPUMP_IOCTL_USBDI_PORT_RESUME_ASYNC	Sent to the USBDI to notify that the function driver resume the port.

Table 32. USBDI IOCTL Notification Codes

Name	Description
USBPUMP_IOCTL_EDGE_USBDI_INSTANCE_ARRIVAL	Sent to notify an instance that a device has arrived and has been bound to a specific instance.
USBPUMP_IOCTL_EDGE_USBDI_INSTANCE_DEPARTURE_ASYNC	Sent to notify an instance that the underlying device has departed. After this IOCTL completes, USBDI will park the instance and will cease using the device. If the device is on a root OTG hub, and HNP is enabled and sensible, then USBDI will attempt to hand host control over to the other device.
USBPUMP_IOCTL_EDGE_USBDI_DEVICE_SUSPEND	Sent to notify an instance that the underlying device has been suspended due to system policies.

8.1 Notification IOCTLs

8.1.1 IOCTLs issued by USBDI to Function Driver instances

The IOCTLS in this section are sent by USBD to function driver instance objects, to notify them of changes in their state.

8.1.1.1 Port reports arrival: `USBPUMP_IOCTL_EDGE_USBDI_INSTANCE_ARRIVAL`

This IOCTL signals the arrival of a device on the indicated port, and that USBD has assigned this instance to handle the device.

Table 33. Fields in `USBPUMP_IOCTL_EDGE_USBDI_INSTANCE_ARRIVAL_ARG`

Field	Description
`USBPUMP_USBDI_INSTANCE_INFO Info;`	Information about the instance, with the following contents:
`USBPUMP_USBDI_PORT *Info.pPort;`	IN: The USBDI port object that will be used for communicating to the device instance.
`USBPUMP_USBDI_PORT_KEY Info.PortKey;`	IN: the key that must be used for all requests issued to the port.
`RECSIZE Info.UrbTotalSize;`	IN: base URB size
`RECSIZE Info.UrbExtraSize;`	IN: required extra bytes in the URB
`RECSIZE Info.HcdRequestSize;`	IN: the portion of the extra bytes that are for the HCD request.
`USBPUMP_USBDI_PIPE_HANDLE Info.hDefaultPipe;`	IN: the handle for the default pipe.
`UINT16 wMaxPower;`	IN: max power in 1mA ticks
`USBPUMP_DEVIDE_SPEED Info.OperatingSpeed;`	IN: the operating speed for this device
`UINT8 Info.bTier;`	IN: tier of this device
`USBPUMP_USBDI_FUNCTION *pFunction;`	IN: The function instance that has been assigned to the device instance.

Architecturally, the class driver is required to save a copy of the Info parameters when this request is received. Otherwise, it will have no way to communicate with the device. However,

to avoid duplication of effort, USBDI will update `pFunction->Function.Info` prior to delivering the notification to the instance's IOCTL manager.

8.1.1.2 Port reports unplug: `USBPUMP_IOCTL_EDGE_USBDI_INSTANCE_DEPARTURE_ASYNC`

This IOCTL signals the departure of a device. When this is received, USBDI has already changed the port key, so no further I/O operations will be permitted. However, it's possible that pending requests have still not completed, although they are in the process of being completed.

Table 34. Fields in `USBPUMP_IOCTL_EDGE_USBDI_INSTANCE_DEPARTURE_ASYNC_ARG`

Field	Description
`USBPUMP_USBDI_PORT *pPort;`	IN: The USBDI port object has been used for communicating to the device instance.
`USBUPMP_USBDI_FUNCTION *pFunction;`	IN: the function instance that was assigned to the device instance.

Be aware that this is an asynchronous IOCTL. Once the function driver completes the IOCTL using `UsbPumpIoctlQe_Complete()`, the code that issued the completion may continue to refer to the instance data only until that function returns control to the DataPump idle loop. Once control passes out of that function, the instance will be released by the central USBDI logic.

Note that UsbPumpIoctlQe_Complete() puts an event on an event queue. In the event that many devices are unplugged at once, this will generate a large flow of system events; one event for each device unplugged. The underlying operating system specifies the maximum number of system events allowed to be queued at one time. This number must be set large enough to accommodate the number of devices used in the system.

8.1.1.3 Port reports suspend/resume: `USBPUMP_IOCTL_EDGE_USBDI_DEVICE_SUSPEND`

This IOCTL signals a change in the suspend/resume state of a device. When this is received, the device state change has already occurred.

Table 35. Fields in `USBPUMP_IOCTL_EDGE_USBDI_INSTANCE_SUSPEND_ARG`

Field	Description
`USBPUMP_USBDI_PORT *pPort;`	IN: The USBDI port object that will be used for communicating to the device instance.
`BOOL fSuspended;`	IN: TRUE if the device is now suspended; FALSE if the device is now active.

Field	Description
USBPUMP_USBDI_FUNCTION *pFunction;	IN: The function instance that has been assigned to the device instance.

Sequence diagrams for device departure are given in Section 9.

8.1.2 Notifications from USBDI to applications

Application-level code may register with USB to get notifications about enumeration success or failure.

To do this, applications must first create a USB DataPump object to represent themselves to USBDI. Then they must open USBDI and register their object with USBDI to receive USBDI enumeration notifications.

Once registered, USBDI will send the following messages to the application object:

Table 36. USBDI IOCTL Enumeration Notification Codes

Name	Description
USBPUMP_IOCTL_EDGE_USBDI_ENUM_FAIL	Sent to indicate that enumeration failed. In an OTG system, there's considerable overlap between this and the USBPUMP_IOCTL_EDGE_OTGCD_ALERT(NotSupported) message, but it happens on any USBDI enumeration attempt. Will be sent for a function on a multi-function device if that function is not recognized.
USBPUMP_IOCTL_EDGE_USBDI_ENUM_ARRIVAL	Sent to indicate that enumeration succeeded; sent for each function on a device.

8.2 Operational IOCTLS

The IOCTLs described in this section are used by Function Drivers to control the operation of USBDI.

8.2.1 Function reports departure: USBPUMP_IOCTL_USBDI_PORT_IDLE_FUNCTION

This IOCTL, which must be sent to the parent port of the function driver instance, tells USBD that the function driver is through working with the device, and is ready to move to the idle state. USBD will subsequently send a departure notification to the function (if it has not already done so). The function driver must wait for the normal departure notification before releasing resources.

Table 37. Fields in `USBPUMP_IOCTL_USBDI_PORT_IDLE_FUNCTION_ARG`

Field	Description
`USBPUMP_USBDI_PORT *pPort;`	IN: The USBDI port object that was used for communicating to the device instance.
`USBPUMP_USBDI_FUNCTION *pFunction;`	IN: pointer to the instance object that is to be idled.

Sequence diagrams for device departure are given in Section 9.

8.2.2 Function reports suspending: `USBPUMP_IOCTL_USBDI_PORT_SUSPEND_FUNCTION`

This IOCTL, which must be sent to the parent port of the function driver instance, tells USBD that the function driver is suspending the port.

Table 38. Fields in `USBPUMP_IOCTL_USBDI_PORT_SUSPEND_FUNCTION_ARG`

Field	Description
`USBPUMP_USBDI_PORT *pPort;`	IN: The USBDI port object that was used for communicating to the device instance.
`USBPUMP_USBDI_FUNCTION *pFunction;`	IN: pointer to the instance object that is to be suspended.

8.2.3 Function reports resuming: `USBPUMP_IOCTL_USBDI_PORT_RESUME_FUNCTION`

This IOCTL, which must be sent to the parent port of the function driver instance, tells USBD that the function driver is resuming the port.

Table 39: . Fields in `USBPUMP_IOCTL_USBDI_PORT_RESUME_FUNCTION_ARG`

Field	Description
`USBPUMP_USBDI_PORT *pPort;`	IN: The USBDI port object that was used for communicating to the device instance.
`USBPUMP_USBDI_FUNCTION *pFunction;`	IN: pointer to the instance object that is to be resumed.

8.2.4 Function requesting OTG port idle or unidle: USBPUMP_IOCTL_USBDI_PORT_SET_IDLE_OTG_PORT

This IOCTL tells USBD that the HCD or PHY driver is requesting to idle or unidle the OTG port. If the port is not an OTG capable port, it will do nothing.

Table 40: Fields in USBPUMP_IOCTL_USBDI_PORT_SET_IDLE_OTG_PORT_ARG

Field	Description
`USBPUMP_USBDI_PORT *pPort;`	IN: The USBDI port object that was used for communicating to the device instance.
`BOOL fSetIdle;`	IN: Flag indicating whether to idle or unidle the OTG port.

9 Device Departure Sequence Diagrams

Figure 6. Instance Gives Up

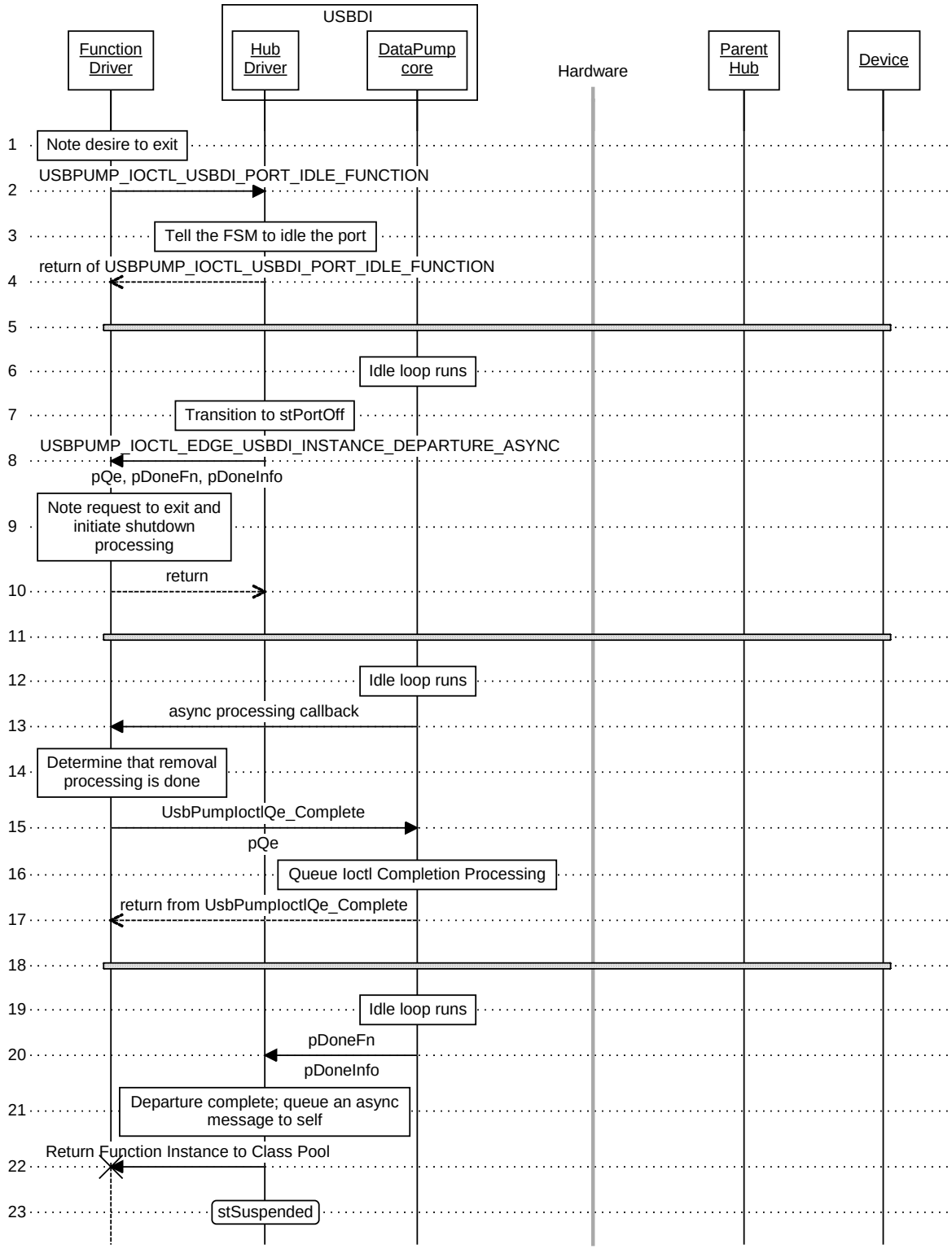


Figure 7. Surprise Removal, typical

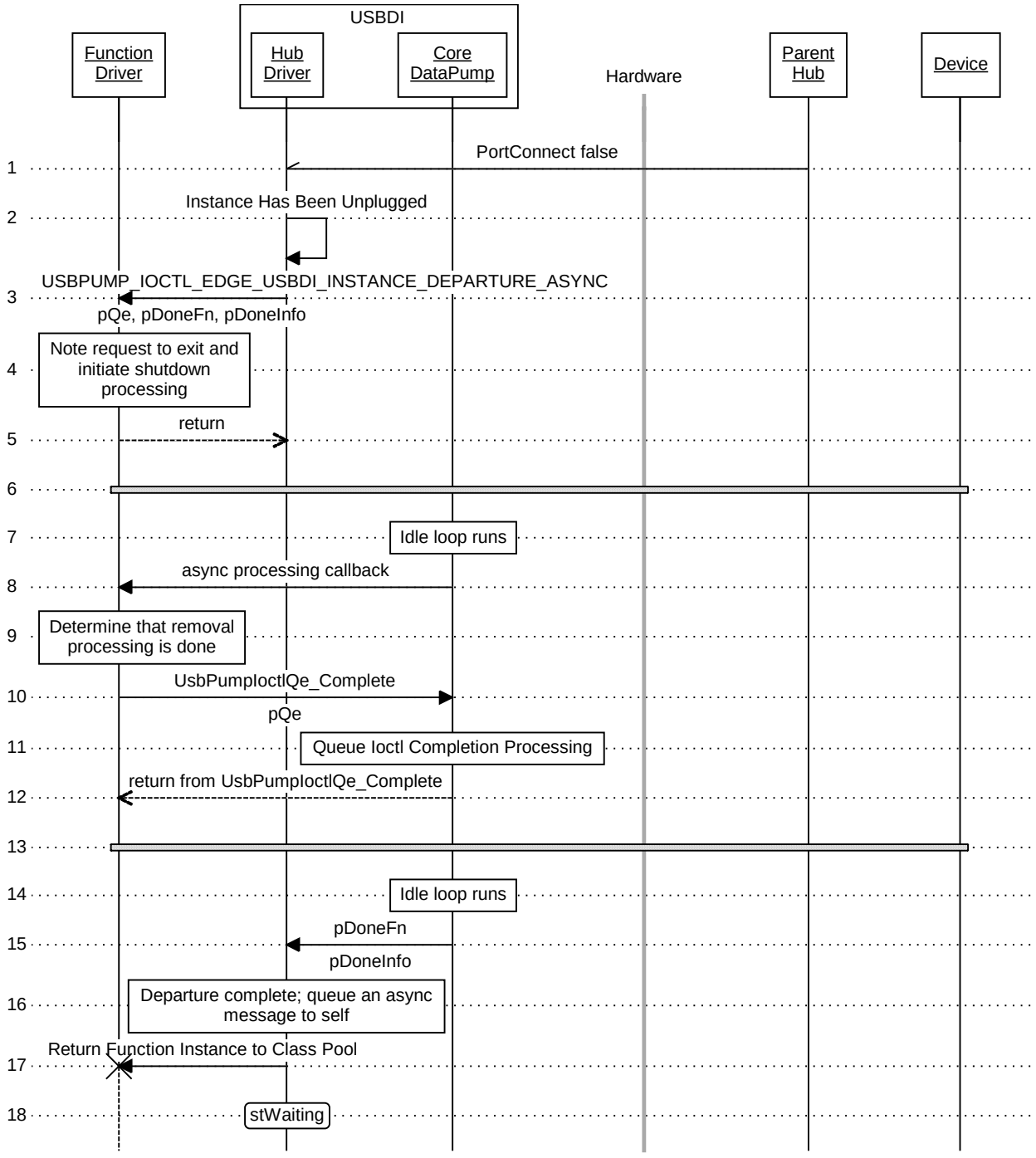
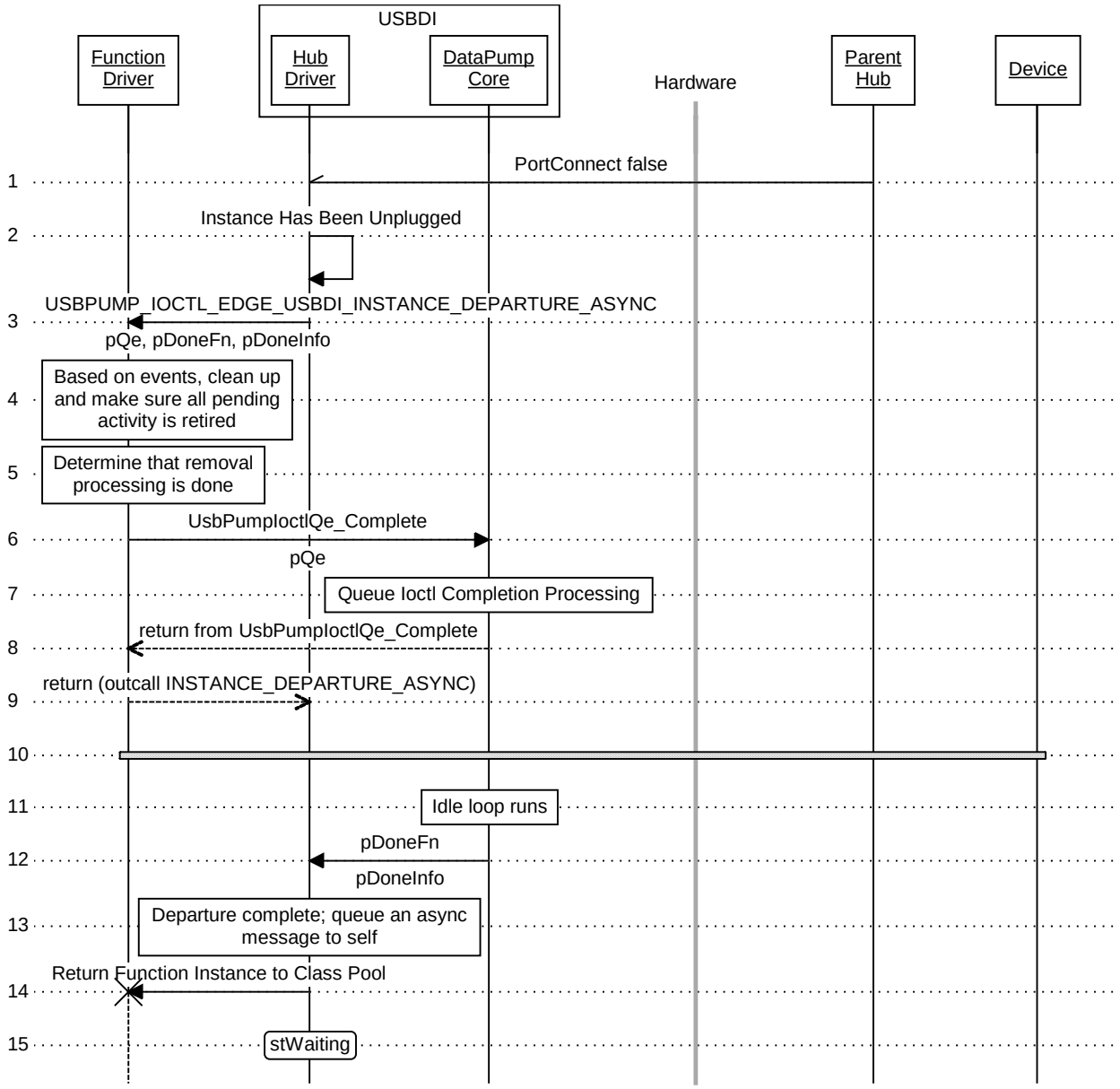


Figure 8. Surprise Removal, Fast



10 Library Code for Function Drivers

10.1 Driver-specific memory pool

```
typedef struct USBPUMP_ABSTRACT_POOL_CONTENTS
        {
        VOID *                      pContext;
        USBPUMP_POOL_ALLOC_FN *     pAllocFn;
        USBPUMP_POOL_REALLOC_FN *   pReallocFn;
```

```
        USBPUMP_POOL_FREE_FN *      pFreeFn;
        USBPUMP_POOL_RESET_FN *     pResetFn;
        USBPUMP_POOL_CLOSE_FN *     pCloseFn;
        USBPUMP_POOL_INFO_FN *      pInfoFn;
        };

typedef union USBPUMP_ABSTRACT_POOL
        {
        USBPUMP_ABSTRACT_POOL_CONTENTS   AbstractPool;
        };
```

Both Driver and port get the required chuncks of memory from the abstract memory pool structure.

10.2 Port-specific memory pools

Both Driver and port get the required chuncks of memory from the abstract memory pool structure (Refer section 10.1).

10.2.1 UsbPumpUsbdPortI_Malloc

```
VOID *
UsbPumpUsbdPortI_Malloc(
        USBPUMP_USBDI_PORT_PRIVATE *pPort,
        BYTES nBytes
        );
```

10.2.2 UsbPumpPlatform_Malloc

```
VOID *
UsbPumpPlatform_Malloc(
        UPLATFORM * pPlatform,
        BYTES nBytes
        );
```

10.2.3 UsbPumpPlatform_MallocZero

```
VOID *
UsbPumpPlatform_MallocZero(
        UPLATFORM * pPlatform,
        BYTES nBytes
        );
```

10.2.4 UsbPumpPlatform_Free

```
VOID
UsbPumpPlatform_Free(
          UPLATFORM * pPlatform,
          VOID * pBuffer,
          BYTES nBuffer
          );
```

10.2.5 UsbPumpUsbdPipeI_Free

```
VOID
UsbPumpUsbdPipeI_Free(
          USBPUMP_USBDI_PIPE *pPipe
          );
```

11 Initializing USBD

```
#include "usbpump_usbdi_api.h"

USBPUMP_OBJECT_HEADER *
UsbPumpUsbd_Initialize(
          USBPUMP_OBJECT_HEADER *pParent,
          CONST VOID *pConfigParam,
          CONST VOID *pImplementationParam
          );
```

Create and return a USBD object. As is usual for standard object creation functions that are to be called from the DataPump abstract initialization layer, this routine returns an object header pointer, rather than a pointer to the USBD object.

`pParent` is used for sending IOCTLs to the rest of the object system, and is used as the dynamic IOCTL parent for USBD. `pConfigParam` is reserved for use as a pointer to a USBD-defined configuration structure, which (if used) is supplied by code external to the DataPump to configure the USBD instance, and which should be a pointer to a `USBPUMP_USBDI_USBD_CONFIG` structure. This is intended to be used to allow users to configure the amount of memory to be used by USBD for things such as fixed buffers. `pImplementationParam` is reserved for use as a pointer to a USBD-defined and –supplied structure of type `USBPUMP_USBDI_USBD_IMPLEMENTATION`; this allows for a variety of MCCI-defined USBD configurations: for example, a configuration that supports full-speed without isochronous, a configuration that supports high-speed with isochronous, and so forth. .

USBD must be initialized after all HCDs have been created for a given system. At present, at most one USBD should be created within a given DataPump instance; however, the system is architected to allow for multiple different USBD implementations that provide radically different methods, e.g. for Wireless USB remote bus proxies.

12 Implementation Code for USBD

The code in this section is documented for reference only, and to assist in understanding the implementation of USBD. None of the functions documented here are to be used outside of USBD. MCCI may revise or replace these functions at any time.

12.1 Initializing USBD

Within USBD, the following function is used to complete initialization started by API functions such as `UsbPumpUsbd_Initialize()`:

```
#include "usbpump_usbd_implementation.h"

USBPUMP_USBDI_USBD_PRIVATE *
UsbPumpUsbdJ_Initialize(
          USBPUMP_OBJECT_HEADER *pParent,
          CONST USBPUMP_USBDI_USBD_CONFIG *pConfigParam,
          CONST USBPUMP_USBDI_USBD_IMPLEMENTATION *pImplementationParam
          );
```

`UsbPumpUsbdJ_Initialize` is NOT an API function, and should not be called directly by code that is external to USBD.

12.2 Generating port keys

```
USBPUMP_USBDI_PORT_KEY
UsbPumpUsbdJ_GeneratePortKey(
          USBPUMP_USBDI_USBD_PRIVATE *pUsbd
          );
```

This function generates a new unique port key from the sequence of port keys issued by the particular USBD instance.

`UsbPumpUsbdJ_GeneratePortKey()` is NOT an API function, and should not be called directly by code that is external to USBD.

12.3 Completing URBs

To complete a URB request that is currently owned by USBDI, USBD functions call:

```
BOOL
UsbPumpUrbI_Complete(
          USBPUMP_URB *pUrb,
          USBPUMP_URB **ppUrbQueue,
```

```
    ARG_USTAT Status
    );
```

`ppUrbQueue`, if not null, points to the queue in which the URB is likely to reside. Even if `ppUrbQueue` is `NULL`, the request will be deleted from any queue it's in. `UsbPumpUrbI_Complete()` returns `TRUE` if and only if the queue element was on a list specified by a non-`NULL` `ppUrbQueue`, and the list is still not empty. (This allows this routine to be used in the same way as `UsbCompleteQe()`.)

Completion is complicated. We first take the request off the specified queue, updating the queue head. Then we reset the cancel routine (and decrement the use count if it was previously set). [If we add support for URB-level timers, we'd further manipulate the reference count here.]

If the reference count ends up at 1, then this routine's job is to complete the request. Based on several considerations, this routine either calls the completion routine directly, or else it transfers the request to USBDI's completion queue (which will cause the request to be completed based on a dispatch from the event loop).

There's a conflict between low latency completions and stack depth. There's also a conflict between low-latency completions and quickly issuing the next request in the queue. There is a throttling mechanism that can be specified when configuring USBDI, so that if too many nested completions occur, the requests are posted to a completion queue.

`UsbPumpUrubI_Complete()` is NOT an API function, and should not be called directly by code that is external to USBD.

12.4 Request Submission Processing

When a request is received, the following steps are taken:

- Verify that the input argument `pDoneFn` is non-`NULL`; otherwise just return.
- Verify that `pPort` and `pRequest` are non-`NULL`; otherwise call `pDoneFn` directly and return.
- Make sure the length is at least the minimum length `(pUrb->Hdr.Length >= sizeof(pUrb->Hdr)` or else call the done function from the input parameter list and give up without touching the request.
- Initialize `Hdr.Status` to busy, `Hdr.Refcount` to zero, `Hdr.pCancelFn` to `NULL`, `Hdr.pCancelInfo` to `NULL`, `Hdr.pUsbdPort` to the `pPort`, `Hdr.pDoneFn`, `Hdr.pDoneInfo` from parameters.
- Since we've set the done function and the done info; from here on, the URB can be completed using the normal path.

- Check that the port key in the URB matches the port key in the port; if not, fail the request.
- Verify that the request code is legal, or fail
- Get the minimum request size from the table.
- Make sure that the length is at least `sizeMin + pPort->Private.nUrbExtraBytes`; otherwise fail the request.
- Verify that the HCD index is greater than `sizeMin`, and less than `pUrb->Hdr.Size – pPort->Private.nUrbExtraBytes`, and that it's aligned; otherwise fail the request.
- Set the internal flags to zero.
- If already dispatching a request for the pipe, queue the request; otherwise increment the request lock and proceed to dispatch.
- Look in the USBDI dispatch table by request; if the function pointer is not `NULL`, call the function pointer.
- Decrement the dispatch count and return.

12.4.1 Processing `USBPUMP_URB_RQ_ABORT_PIPE`

For all local ports, the process is the same: set the pipe-halted flag. Then walk the list of requests that are pending at the HCD, and issue the HCD cancel request. Then walk every request that is waiting in the upper queue, and issue a URB cancel.

12.4.2 Processing `USBPUMP_URB_RQ_BULKINT_IN`

Convert the pipe handle to a pointer to a `USBPUMP_HCD_PIPE` & verify type & direction.

If the pipe is halted, bounce the request

Prepare an appropriate HCD request using the spare bytes, and send it down

On completion, update the URB status from the HCD status

12.4.3 Processing `USBPUMP_URB_RQ_BULKINT_OUT`

Convert the pipe handle to a pointer to a `USBPUMP_HCD_PIPE` & verify type & direction.

If the pipe is halted, bounce the request

Prepare an appropriate HCD request using the spare bytes, and send it down

On completion, update the URB status from the HCD status

12.4.4 Processing USBPUMP_URB_RQ_CONTROL_IN

Convert the pipe handle to a pointer to a USBPUMP_HCD_PIPE & verify type & direction.

If this is the default pipe, filter the standard commands

[Composite device ports will use code based on the MCCI bus driver to multiplex default pipe operations from different devices.]

On completion, update the URB status from the HCD status.

12.4.5 Processing USBPUMP_URB_RQ_CONTROL_OUT

Convert the pipe handle to a pointer to a USBPUMP_HCD_PIPE & verify type & direction.

If this is the default pipe, filter the standard commands. Note that Root ports let a few more things through (SET_ADDRESS, in particular)

[Composite device ports will use code based on the MCCI bus driver to multiplex default pipe operations from different devices.]

On completion, update the URB status from the HCD status.

12.4.6 Processing USBPUMP_URB_RQ_DEFINE_CONFIG

Initiate processing of a USBPUMP_URB_RQ_DEFINE_CONFIG URB after preliminary verification of the URB has been completed. Implementing the request involves reserving bandwidth and power, but doesn't require doing any I/O -- so we just hog the USB thread while we perform our calculations. This involves four steps.

- Check to make sure it OK to change the config tree.
- Undefine the current configuration if there is one.
- Define the new configuration.
- Complete the URB.

12.4.7 Processing USBPUMP_URB_RQ_GET_DEVICE_INFO

Process USBPUMP_URB_RQ_GET_DEVICE_INFO to request USBD to fill in information about the resources being consumed by the device (or function, in case this is a composite device).

12.4.8 Processing `USBPUMP_URB_RQ_GET_FRAME`

Prepare a request USBPUMP_HCD_RQ_GET_FRAME, and send it down. On completion, copy the data from the HCD request to the URB and complete the URB.

Builds an HCD request and dispatches it to the HCD handler. This request asks USBD to return the current frame number or microframe number for the bus on which the port resides.

12.4.9 Processing `USBPUMP_URB_RQ_GET_PORT_STATUS`

Process USBPUMP_URB_RQ_GET_PORT_STATUS to request USBD to return the current status of the port.

12.4.10 Processing `USBPUMP_URB_RQ_ISOCH_IN`

Convert the pipe handle to a pointer to a `USBPUMP_HCD_PIPE` & verify type & direction.

If the pipe is halted, bounce the request

Prepare an appropriate HCD request using the spare bytes, and send it down

On completion, update the URB status from the HCD status

12.4.11 Processing `USBPUMP_URB_RQ_ISOCH_OUT`

Convert the pipe handle to a pointer to a `USBPUMP_HCD_PIPE` & verify type & direction.

If the pipe is halted, bounce the request

Prepare an appropriate HCD request using the spare bytes, and send it down

On completion, update the URB status from the HCD status

12.4.12 Processing `USBPUMP_URB_RQ_REENUMERATE_PORT`

Prepare a USBPUMP_URB_RQ_REENUMERATE_PORT request, which is used to force USBDI to reenumerate a port.

12.4.13 Processing `USBPUMP_URB_RQ_RESET_PIPE`

Pass a reset pipe request down to the HCD layer. This request causes USBDI to reset the specified pipe. Any halt status in the pipe is reset. This request only resets the endpoint-halted flag in the pipe database. It does not reset the data toggles. Therefore, this request is rarely used. To reset the data toggle and clear the endpoint-halted flag, use USBPUMP_URB_RQ_CONTROL_OUT to send CLEAR_FEATURE(ENDPOINT_HALT).

12.4.14 Processing `USBPUMP_URB_RQ_RESET_PORT`

Call PortFsm to indicate ResetPort was requested. Check port pointers. If NULL, complete the URB with error; otherwise, call the PortFsm.

13 USBDI Configuration

When USBDI is initialized, the caller provides two pointers. These two pointers determine the capabilities that will be offered by the specific USBDI implementation. These pointers are known as the configuration pointer and the implementation pointer.

The implementation pointer is an opaque pointer to an MCCI-supplied object. This object determines (at link time) the capabilities of the USBDI implementation, including:

- Presence or absence of high-speed support in USBDI
- Presence or absence of isochronous support in USBDI
- Presence or absence of transaction-translator support in USBDI
- The default name to be used for the USBDI instance

The configuration pointer points to a user-supplied USBPUMP_USBDI_USBD_CONFIG object. This object specifies a number of run-time-configurable parameters such as:

- The maximum supported configuration descriptor size (during enumeration)
- The name to be used for the USBDI instance (overriding the built-in default name).

13.1 The USBPUMP_USBDI_USBD_CONFIG Object

```
typedef struct USBPUMP_USBDI_USBD_CONFIG
        {
        UINT32                                  MagicBegin;
        BYTES                                   Size;
        CONST TEXT *                            pUsbdName;
        BYTES                                   maxNestedCompletions;
        UINT16                                  sizeConfigBuffer;
        UINT16                                  sizeStringDescBuffer;
        BYTES                                   maxUrbExtraBytes;
        UINT16                                  tAttachDebounce;
        UINT16                                  tResetRecovery;
        UINT16                                  tSetAddrCompletion;
        UINT16                                  tSetAddrRecovery;
        UINT16                                  tStdRequestNoData;
        UINT16                                  tStdRequestData1;
        UINT16                                  tStdRequestDataN;
        UINT16                                  tStdRequestMinimum;
```

```
        UINT16                                  tResumeRecovery;
        UINT8                                   bNumberHubs;
        UINT8                                   bPortsPerHub;
        UINT16                                  tHostInitiatedResumeDuration;
        CONST USBPUMP_USBDI_INIT_MATCH_LIST *   pHubIdOverrides;
        UINT                                    AnnunciatorMaxSession;
        UINT32                                  ulDebugFlags;
        UINT32                                  MagicEnd;
        };
```

13.2 The USBPUMP_USBDI_USBD_IMPLEMENTATION Object

```
typedef struct USBPUMP_USBDI_USBD_IMPLEMENTATION
        {
        UINT32                              MagicBegin;
        BYTES                               Size;
        BYTES                               sizeUsbd;
        CONST TEXT *                        pUsbdName;
        USBPUMP_OBJECT_IOCTL_FN *           pIoctlFn;
        CONST USBPUMP_URB_DISPATCH_TABLE *  pUrbDispatch;
        UINT32                              MagicEnd;
        };
```

13.3 Pre-defined Implementations

13.3.1 Minimal Implementation: `gk_UsbPumpUsbdImplementation_Minimal`

This implementation object defines a USBD implementation that includes support for full- and low-speed control transfers, interrupt transfers, and interrupt transfers, but no support for isochronous or high-speed transfers.

14 Open Implementation Questions

On composite devices, will the USBPUMP_USBDI_PORT object implementation routines be different than they are for real devices?

15 Scheduling

Scheduling is done with two layers of code.

The first layer is an abstract scheduler, and is part of USBDI. This layer keeps track of the bandwidth that has been booked using USBPUMP_URB_RQ_DEFINE_CONFIG, by building a schedule tree representing the existing reservations for periodic traffic. This layer is implemented using a tree of bandwidth counters; for full-speed busses, the tree is preconfigured to have 32 periodic buckets representing periodic traffic that is once every 32 ms,

and then a sequence of halvings, each time representing possible start times for periodic traffic that's twice as often: 16 buckets representing the periodic traffic that is every 16 ms, 8 buckets representing traffic that is every 8 ms, 4 buckets representing traffic that is every 4 ms, 2 buckets representing traffic that is every 2 ms, and one bucket representing the traffic that is scheduled every frame. Thus, a vector of 63 integers is sufficient to represent the bandwidth allocation for the bus. By convention, the root node includes the sum of its two children; these in turn include the sums of their children, and so forth. When adding a periodic transfer to the list, one must add the bandwidth to its node of the tree, and then propagate the changes towards the root.

The second layer is a physical scheduler; this is part of the HCD. HCD Pipes are (conceptually) entered into the schedule by the `USBPUMP_HCD_RQ_INIT_PIPE` operation, and are taken out by the `USBPUMP_HCD_DEINIT_PIPE` operation. USBD is supposed to be smart enough only to initialize pipes that will "fit" in the schedule.

Each periodic pipe data structure must incorporate information that lets our scheduling code work.

MCCI's USBD takes an unusual approach to allocating bandwidth: we allocate it for an entire range of possibilities, not just for a single possibility. This simplifies the problem of creating a configuration, but complicates the problem of booking bandwidth.

We handle this by creating a second (work) tree during scheduling. This second tree contains the deltas at each level (either for addition or subtraction). The memory for this tree is allocated in the per-bus object.

15.1 Processing DEFINE_CONFIG

The processing of DEFINE_CONFIG proceeds as follows. Reset all the mapping of periodic pipes, and if there are none, go on to checking power.

Otherwise, we need to create a max-sum tree for the DEFINE_CONFIG, and we will need a max-sum tree for tracking a single view of the device.

We have to have a max-sum tree for the device, and a max-sum tree for the current view of the device. For each view of the device, we have to build the max-sum tree (and map the pipes if they're not already mapped) minimizing the sum of the base tree, the bandwidth already assigned in this view, and the bandwidth consumed by this pipe.

Logically, what we need to do is build a family of trees, one for each possible view of the device; showing the delta from the containing view of the device; and then we need to iterate over each of those views, and at each level in the tree record the maximum amount of resource that will be consumed by applying that tree.

Furthermore, this tree family has to be expressed in terms of the base (in the bus) without the previous view of the device applied.

We also need to be able to reconstruct the view of the bandwidth consumed by a previous DEFINE_CONFIG, so we can throw it away (when DEFINE_CONFIG [NULL] happens). So we have to store the tree.

1. Initialize the work bandwidth-mapping tree to zero.

2. For each configuration: Make a pass over the pipes in the configuration. Process the pipes that are in interfaces without alternate settings, and add their bandwidth to the work table (and map them to slots). Then assign each periodic pipe to a slot in the work tree (setting the level and slot variables, and adding the bandwidth consumed by the endpoint to the slot).