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 Host Controller Driver API

Engineering Report 950000324
Rev. F
Date: 2011/09/28

PROPRIETARY NOTICE AND DISCLAIMER

Unless noted otherwise, this document and the information herein disclosed are proprietary to MCCI Corporation, 3520 Krums Corners Road, Ithaca, New York 14850 ("MCCI"). Any person or entity to whom this document is furnished or having possession thereof, by acceptance, assumes custody thereof and agrees that the document is given in confidence and will not be copied or reproduced in whole or in part, nor used or revealed to any person in any manner except to meet the purposes for which it was delivered. Additional rights and obligations regarding this document and its contents may be defined by a separate written agreement with MCCI, and if so, such separate written agreement shall be controlling.

The information in this document is subject to change without notice, and should not be construed as a commitment by MCCI. Although MCCI will make every effort to inform users of substantive errors, MCCI disclaims all liability for any loss or damage resulting from the use of this manual or any software described herein, including without limitation contingent, special, or incidental liability.

MCCI, TrueCard, TrueTask, MCCI Catena, and MCCI USB DataPump are registered trademarks of MCCI Corporation.

MCCI Instant RS-232, MCCI Wombat and InstallRight Pro are trademarks of MCCI Corporation.

All other trademarks and registered trademarks are owned by the respective holders of the trademarks or registered trademarks.

NOTE: The code sections presented in this document are intended to be a facilitator in understanding the technical details. They are for illustration purposes only, the actual source code may differ from the one presented in this document.

NOTE: The code sections presented in this document are intended to be a facilitator in understanding the technical details. They are for illustration purposes only, the actual source code may differ from the one presented in this document.

Document Release History

Rev. A	2007/07/08	Original release
Rev. B	2007/08/17	Add info on HCD_PIPE.
Rev. C	2007/12/21	Add more IOCTLs
Rev. D	2008/12/18	Correct diagrams
Rev. E	2011/09/05	Changed all references to Moore Computer Consultants, Inc. to MCCI Corporation. Changed document numbers to nine digit versions.

Rev. F	2011/09/28	Added source code disclaimer.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 Introduction

1.1 Purpose

This document specifies the APIs for Host Controller Drivers (HCDs) that are part of the MCCI DataPump embedded host (EH) and USB On The Go (OTG) product.

Because the HCD may need to integrate the OTG control and transceiver control functions, it makes sense to document those features here. However, these functions may also be provided as separate functions, depending on how the hardware is composed.

1.2 Scope

This document contains tracking and background information for the development phase of this project. It will become obsolete at the end of the development phase, when it will be replaced by user technical documentation.

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].
Carkit	A carkit is an after-market device that is installed in a car. It contains a speaker and a microphone, and draws current from the car power adapter. A carkit acts as a speaker-phone attachment to a cell phone, and allows hands free operation of a cell phone.
DCD	*See* Device Controller Driver
Device controller	The hardware module responsible for connecting a USB device to the USB bus.
Device Controller	The software component that provides low-level access to the specific Device Controller in use. All MCCI USB DataPump DCDs implement a common

Driver (DCD)	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
HcdKit	The MCCI standard HCD driver framework
HCD Class	*See* Host Controller Driver
HCD Instance	*See* Host Controller Driver
HNP	Host Negotiation Protocol
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 shorthand for "transceiver". MCCI uses this in the abbreviations for the API operations that are used for accessing the phy.
SRP	Session Request Protocol
Transceiver	The hardware module responsible for low-level signaling on the USB bus.
USBPHY	USB Transceiver Driver
xCD	Host, Device, Dual-Role or OTG Controller Driver

1.4 Referenced Documents

[CEA936] *CEA Standard CEA-936, Mini USB Analog Carkit Interface Specification*, Consumer Electronics Association, December 2002. Available from Global Engineering Documents, **http://global.ihs.com**.

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

[ISP1301] *ISP1301Universal Serial Bus On-The-Go Transceiver Product Data*, Philips Semiconductor, Document Order Number 9397 750 14337, released 2005-01-04.

[MEHDEF] *MCCI DataPump EH and OTG Project Definition*, MCCI Engineering Report 950000330

[MUSBDI] *MCCI USB DataPump Embedded USBDI*, MCCI Engineering Report 950000325.

[USBBOT10] *Universal Serial Bus Mass Storage Class Bulk Only Transport,* version 1.0 (also referred to as the *BOT Specificatio*n. This specification is available at **http://www.usb.org/developers/devclass_docs.**

[USBCBI11] *Universal Serial Bus Control/Bulk/Interrupt (CBI) Transport*, version 1.1 (also referred to as the *CBI Specification*. The specification is available at **http://www.usb.org/developers/devclass_docs.**

[USBCORE] *Universal Serial Bus Specificatio*n, version 2.0/3.0 (also referred to as the *USB Specification*), with published errata and ECOs. This specification is available on the World Wide Web site **http://www.usb.org**.

[USBMSC14] *Universal Serial Bus Mass Storage Class Specification Overview,* version 1.4 (also referred to as the *MSC Specification*). This specification is available at **http://www.usb.org/developers/devclass_docs.**

[USBOTG20] *On-The-Go and Embedded Host Supplement to the USB 2.0 Specification*, version 2.0 (also referred to as the *OTG Specification*). This specification is available at (**http://www.usb.org/developers/onthego**).

2 Overview

The MCCI USB DataPump HCD API is a portable framework for developing host controller drivers for the MCCI USB DataPump Embedded Host / OTG environment.

The design has the following goals:

1. Keep the hardware-specific code as simple as possible

2. Don't mix USBD concepts into the HCD layer.

3. Leave bus bandwidth allocation and other messy details to USBD

4. Provide for common handling of basic HCD plumbing from a common library.

5. Allow flexibility in case the common library architecture is not sufficient to our needs.

6. Allow for unit-testing of the HCD components

In addition to specifying a client API for use by USBD and by specialized unit test applications, this document also specifies a generalized HCD framework, called "HcdKit".

3 Summary of Layers

The following several figures show how the layers from HCD and DCD to hardware are arranged for various configurations. Note that several of the layers are effectively bypassed except for control operations.

- Figure QQQ shows a simple device-only configuration

- Figure QQQ shows a host-only configuration

- Figure QQQ shows an OTG dual-role device

- Figure QQQ shows the configuration that is used for Catena 1620, which has an ISP1362. The ISP1362 has 2 ports and four operating modes; the Catena 1620 allows any of the following five modes to be chosen by attaching the proper connector:

  Device only

  Dual-port host only

  Host + device

  OTG dual-role device

  Host + OTG dual-role device

- Figure QQQ shows the configuration that is used for a hypothetical product that can dynamically switch its single port as follows:

  a) Device

  b) Host

  c) RS232 for accessories

  d) Car kit mode

Note that OTG is not supported; presumably the mode switch is handled by a combination of cables and UI. (A device that had a series of proprietary adapter cables could realize this design.)

From these variations, it can be seen that:

1. The transceiver layer is responsible for knowing how the transceiver can be configured, and how it is in fact configured.

2. If the transceiver is programmable under software control, the transceiver layer must provide APIs for switching modes. (This API is provided by IOCTL operations, and therefore is visible by external processes by sending the appropriate messages to the DataPump.)

3. The HCD, DCD and OTG policy modules cannot talk to the transceiver directly; instead they must use a separate set of IOCTL operations. Some of the protocol requests (i.e., SRP, HNP, etc) are necessarily implemented by code provided by the xCD, but

4. The OTG layer, if present, operates by talking to the transceiver layer, and must be prepared for the transceiver layer to be in use for another (non USB) purpose.

5. The DCD and HCD layers must be prepared for logical disconnects from the transceiver; but this is not a problem in general, because this can be modeled as a cable disconnect.

4 USB Transceiver (Phy) interface

All USB devices and hosts are (at least logically) connected to the bus port using USB transceivers. These transceivers, being physical port pins, may have programmable modes. For traditional USB 1.1 Device-only systems, the transceiver was normally integrated into the baseband device controller chip, and equivalent functionality was incorporated into the DataPump DCD model. However, with the advent of high speed USB, USB car kits, USB On-The-Go, and numerous vendor-specific multiplexing approaches, the MCCI USB DataPump version 3.0 includes APIs that allow clients to access the transceiver in a consistent way.

Control communications with the transceiver may be done in several ways: for example, through dedicated registers in the HC hardware; through GPIO pins; through a separate I²C controller; and so on. In the case of a USB device, the transceiver is usually co-located with the device controller, and the DCD works directly with it, possibly using subroutines that are integrated in the DCD's "wiring" configuration structure.

USB hosts and devices are inherently asymmetric in several ways. This is true in the case of transceiver location as well. Embedded host (and therefore OTG) controllers often use transceiver chips such as the Philips ISP1301, which have separate, I²C control interfaces. Depending on the HC core in use, the I²C controller might reside in the HC, might reside elsewhere in the system, or might have to be implemented by software using GPIO pins. The I²C bus might be shared with other uses, and therefore might be busy when the HCD needs to access it.

Furthermore, the transceiver can be used for non-USB purposes, such as accessing an external car kit, accessing other external accessories.

Transceiver management is therefore potentially a big issue. However, for the purposes of this discussion it suffices to say that new DataPump HCDs and DCDs are required to export a transceiver API defined by the DataPump. This API models the Transceiver as a separate object. However, since the Transceiver is normally detected by the xCD, the xCD is responsible for creating this object as part of its attach sequence.

The transceiver layer has the following functions. All are asynch, to allow for remote calls:

USBPUMP_IOCTL_PHY_GET_INFO_ASYNC	Get a property from the transceiver
USBPUMP_IOCTL_PHY_SET_INFO_ASYNC	Sets a property to the transceiver

5 Initialization and Top Level Representation

Any MCCI USB DataPump EH or OTG implementation starts out by having one or more HCD Classes referenced from the initialization vector of the DataPump.

5.1 Initialization

By convention, all HCDs should export a function with the following signature:

```
typedef PUSBPUMP_HCD
USBPUMP_HCD_INIT_FN(
          USBPUMP_OBJECT_HEADER *pParent,
          CONST VOID *pInitParam
          );
```

The parameter `pInitParam` is intended to be used to pass wiring information as defined by the HCD code from the DataPump configuration system to the HCD's initialization functions. `pParent` is the object that is to be the IOCTL parent for this object; normally it is the platform object, but it need not be. The platform and root pointers may be obtained by calling `UsbPumpObject_GetPlatform(pParent)` and `UsbPumpObject_GetRoot(pParent)`, respectively.

To initialize an HCDKIT HCD, the USBPUMP_HCD_INIT_FN implementation should first allocate a memory buffer of the appropriate size for the overall HCD. It then should call:

```
BOOL
UsbPumpHcdKit_InitializeHcd_V1(
          USBPUMP_HCDKIT_HCD *pHcd,
          SIZE_T HcdSize,
          USBPUMP_OBJECT_HEADER *pParent,
          CONST TEXT *pHcdName,
          CONST USBPUMP_HCDKIT_SWITCH *pSwitch,
```

```
    BYTES nRootPorts,
    USBPUMP_DEVICE_SPEED_MASK SupportedSpeedMask,
    UINT32 InitialStatus,
    BYTES DefaultMaxTransferSize
    );
```

This is an explicitly versioned API; after development freeze, MCCI will not revise this API, but will provide new versions if updates are needed, and will revise this API to provide the appropriate parameters to the new API.

Comments on parameters follow:

- `pHcd` should point to the actual HCD instance data object. `HcdSize` specifies the actual size. This must be at least as large as `sizeof(USBPUMP_HCDKIT_HCD)`. The `USBPUMP_HCDKIT_HCD` prefix is zeroed, and then properly initialized and linked into the object system. However, the tail (between `sizeof(USBPUMP_HCDKIT_HCD)` and `HcdSize`) is not overwritten.

- The name passed as `pHcdName` should be generated using the macro `USBPUMP_HCD_NAME()`, which will generate a well-formed HCD name by appending "`.hcd.mcci.com`" to the string.

- `pParent` points to the object that will be the IOCTL parent of the new HCD instance. Typically, this is the PHY object.

- `pHcdName` gives the instance name that will be given to this object, and should be constructed using the macro `USBPUMP_HCD_NAME("xx")`.

- `pSwitch` points to a table of functions that provide the virtual methods for this HcdKit object.

- `nRootPorts` gives the number of ports on the root hub, and must be greater than zero.

- SupportedSpeedMask gives the supported speed mask of the HCD. It should normally be combination of USBPUMP_DEVICE_SPEED_MASK_FULL, USBPUMP_DEVICE_SPEED_MASK_HIGH or USBPUMP_DEVICE_SPEED_MASK_SUPER.

- `InitialStatus` gives the initial status of the HCD (in particular, whether it's up or down).

- `DefaultMaxTransferSize` gives the max transfer size that is to be used as the default for this HCD, if none is specified by the client.

Each HCD has a class meta description. This description includes a pointer to an initialization function; the meta-description is intended to be embedded in a larger object class-specific structure.

HCDs are created in two steps.

First, the HCD class object must be created, by calling UsbPump_Init(), using the root object as the superclass. This will be named "isp1362.hcd.driver.mcci.com", etc.

The class object's IOCTL methods must include a method, USBPUMP_IOCTL_CLASS_INIT_INSTANCE, which will initialize an instance, given a block of memory that is large enough to hold the instance. This will including setting that instance's

The initialization function is always called the following way:

```
// create the class
pClassObjectHdr = xxx_

// if needed . . .
pClassObjectHdr = UsbPumpObjectFirstMatch(
                      pIoctlParent,
                      "isp1362.hcd.driver.mcci.com"
                      );

pObject = UsbPumpObject_CreateFromClass(
              pClassObjectHdr,
              pIoctlParent,
              pParams
              );
```

- pClassObjectHdr points to an object that will accept USBPUMP_IOCTL_GET_CLASS_METHODS IOCTLs. This class object will be the class parent of the created object.

- pIoctlParent points to the object that is the dynamic parent for the set of objects created by this method. Normally, the created internal objects are stacked between pParent and the final object, but this is determined by the intialization method

- pParams points to optional data – it's passed into the GET_CLASS_METHODS implementation. However, it's derived from type USBPUMP_CLASS_INIT_DATA, which means that it starts with a length.

Although the creation operation always returns a single object pointer, it may create as many objects as are required. Its only constraint is that it must return a single object pointer.

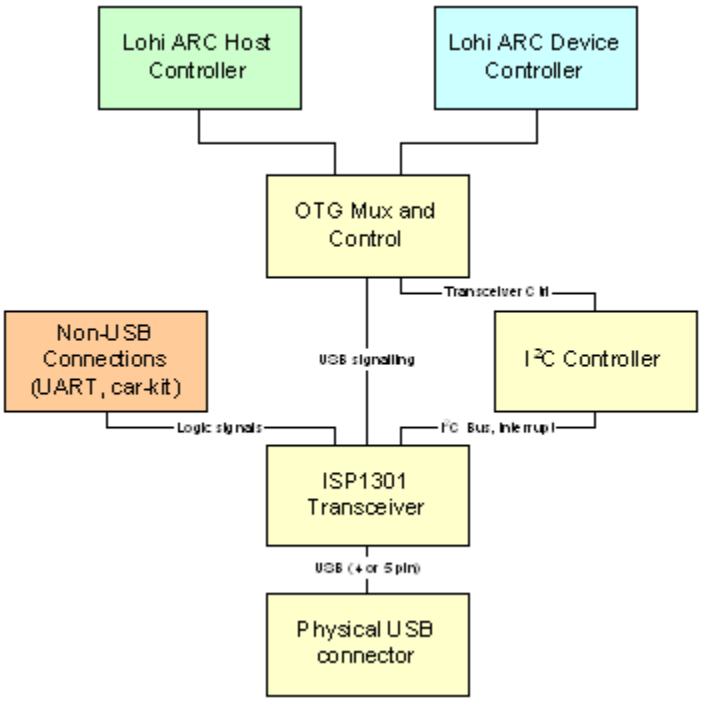
5.2 Some scenarios

For illustrative purposes, we consider hardware and related software data structures for a variety of USB HCD/DCD combinations.

5.2.1 Lohi ARC core

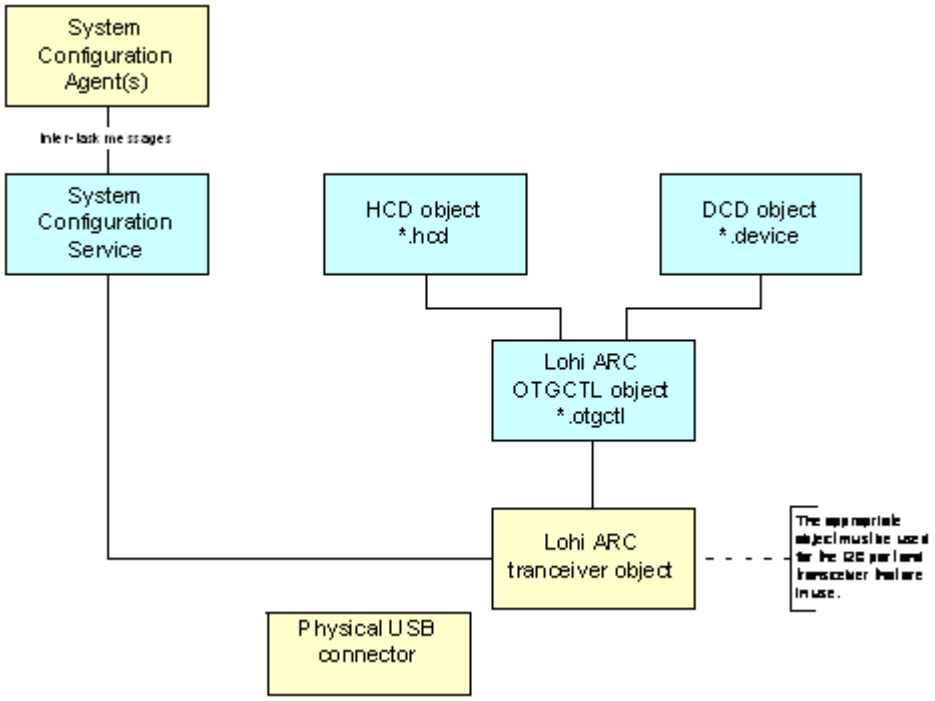
The UPLATFORM provides the phy, and there's only one. However, the phy is controlled via IOCTLs, which are sent down to the UPLATFORM via the UDEVICE. So we can intercept this between the UDEVICE and the UPLATFORM.

Figure 1. Lohi Hardware Block Diagram



The software model is:

Figure 2. Lohi Software Block Diagram



During device driver initialization, given an input pointer to the UPLATFORM object, we will need to create abstraction points for:

- The transceiver – the current device-only driver is handling this directly, but will have to share. Since the phy is external, this has to be separate code from the OTG control object.

  This layer needs to get port power capability, etc, from somewhere. This should include the port's existence, power capabilities, etc. Of course, this can be initialized "by hand" on the first build.

- The OTG control, which needs to include all the central register initialization for the device

- The device, as child of the OTG control object – this needs the info from app init, including a pointer to the app init code

- The host, as child of the OTG control object

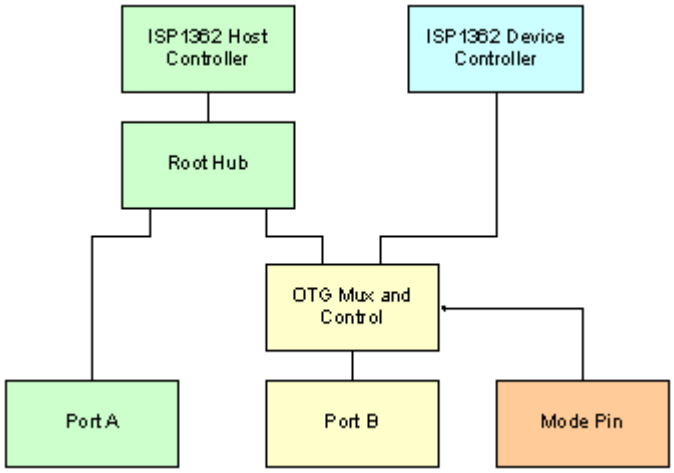
Right now, the transceiver layer and OTG control are buried in the platform; they need to be pulled out.

To minimize memory on dynamic restarts, the user may need to write a wrapper for the transceiver logic which fails initialization (doesn't create the transceiver object) if MMI or other considerations indicate that the USB subsystem should not be operated.

5.2.2 Philips ISP1362

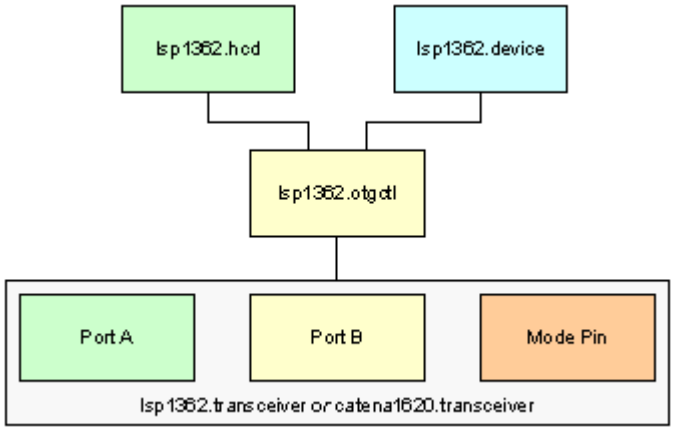
The ISP1362 itself provides the phy, and there's only one. However, there are multiple configurations; and in the Catena the configuration is not pre-determined. The device has the following layout:

Figure 3. ISP1362 Hardware Block Diagram



The software model is:

Figure 4. ISP1362 Software Block Diagram



- The hardware (transceiver/wiring) has to indicate the possible configurations:

  a) Completely disconnected

  b) Port A is physically present or not, & power availability

c) Port B is physically present or not & power availability

d) Power configuration and capacity

- During initialization, the HCD learns about the capabilities of the root hub by sending IOCTLs to the lower object. These IOCTLs are 1362-specific. It's a hardware fact that changing the cable will change whether there is an attach signaled at the port, so there is no reason to build cable awareness into the HCD (apart from power management issues).

- The HCD might refuse to create itself if during initialization it learns that Port A is not physically present, and that Port B is configured as a device port.

- During initialization, the DCD should refuse to create if the transceiver indicates that no device is every possible (i.e., port B is not wired)

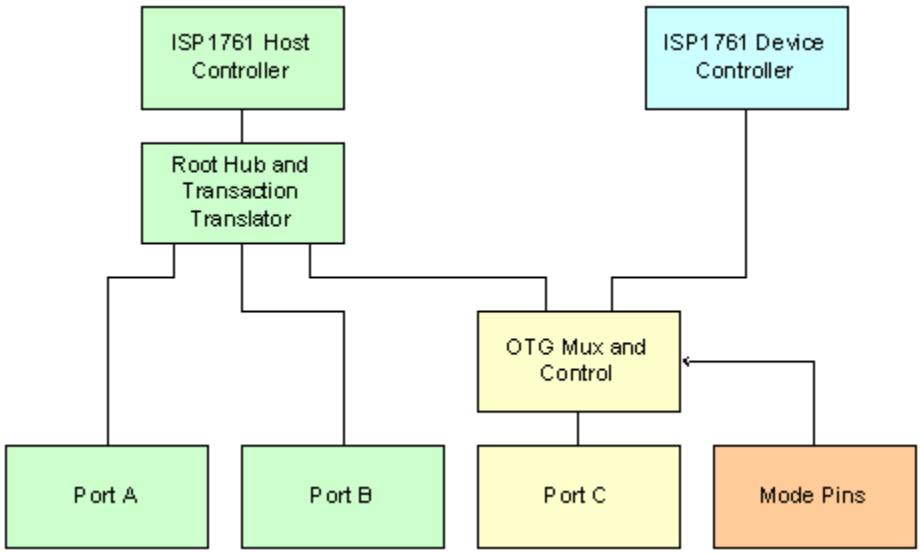
- The DCD's API usage for the OTGCTL layer is simply that it may ask for SRP as a wakeup

5.2.3 Philips ISP1761/2

The ISP1761 itself supplies the phy, and there's only one. However, there are three ports; and there's also high-speed support.

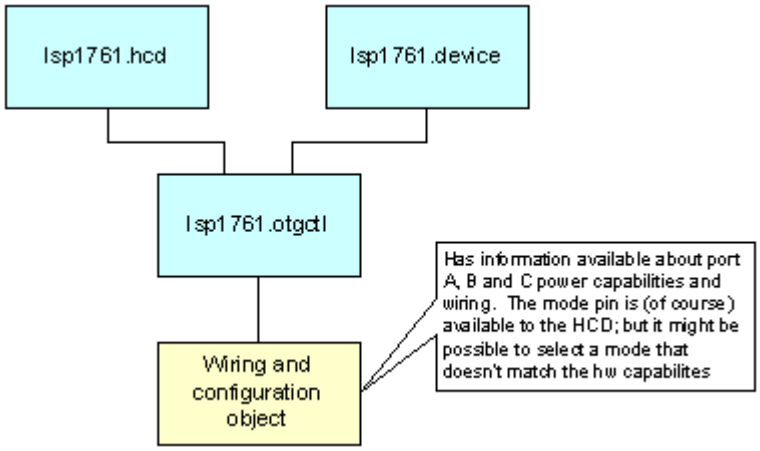
The ISP1362 itself provides the phy, and there's only one. However, there are multiple configurations; and in the Catena the configuration is not pre-determined. The device has the following layout:

Figure 5. ISP1761 Hardware Block Diagram



The software model is:

Figure 6. ISP1761 Software Block Diagram



5.2.4 Mentor Core

The Mentor core adds the additional wrinkle that it can be used either with a FS-only PHY or a HS phy. So the phy API needs (possibly) to indicate whether high-speed is possible; this will at least affect correct processing of the test commands, as well as of the DEVICE_QUALIFIER and OTHER_SPEED_CONFIG descriptor requests.

6 Handling Common Scenarios

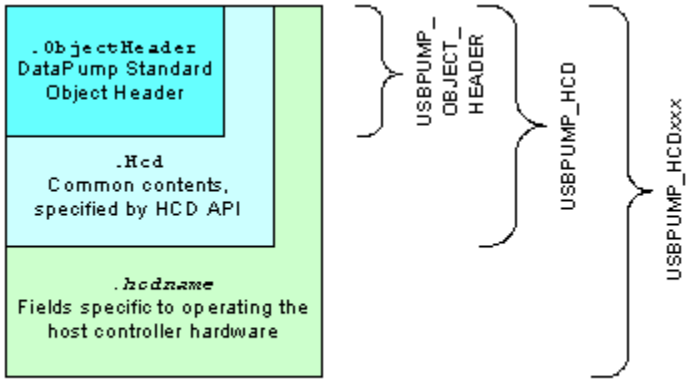
Many operational scenarios for an HCD can be simulated to the USBDI layer by manipulating the root hub. For example, the root hub request for status can simply be completed to indicate removal of all devices. However, the root hub descriptor cannot be read before hardware arrives; and similarly the registers of the root hub cannot be read after the hardware departs. This means that USBDI will be able to launch the root hub manager based on hardware arrival, and to stop the root hub queries on hardware departure.

Because of power management, we assume that all HCDs are (effectively) removable. The hardware must reject all requests received when down, and complete (abort) all pending requests when it receives a "down" notification.

7 Common HCD Object Contents

The HCD API is organized by the contents of the common HCD object. The common HCD object allows common services that are not hardware-specific to be offered by the DataPump library. Therefore, the HCD layout is composed of at least three parts, overlaid in memory using a hierarchy of C union and struct types. The layout is shown in Figure 7. These parts may be considered to be hierarchical views of the same block of data, ranging from most to least abstract.

Figure 7. Full HCD Driver USBPUMP_HCD Object Layout



The most abstract view of the USBPUMP_HCD is as a USB DataPump object. In this view, it is exactly the same as any other objects. The object header fields are accessed using the notation:

```
pObj->ObjectHeader.FieldName
```

The next less abstract view is the USBPUMP_HCD. This structure starts with the same USBPUMP_OBJECT_HEADER field as before, but adds new fields with the prefix "`Hcd`". So for example:

```
USBPUMP_HCD *pHcd;
. . .
if (pHcd->ObjectHeader.InstanceNumber = 0)
       /* do something based on this being the first instance... */

. . .

(*pHcd->Hcd.InSwitch.pSubmitRequest)(
       pHcd,
       pHcdRequest,
       pDoneFn,
       pDoneInfo
       );
```

8 Full HCD Drivers and HcdKit

From the client's point of view, the HCD layer exports one API, with several access methods:

1. Access via DataPump IOCTLs.

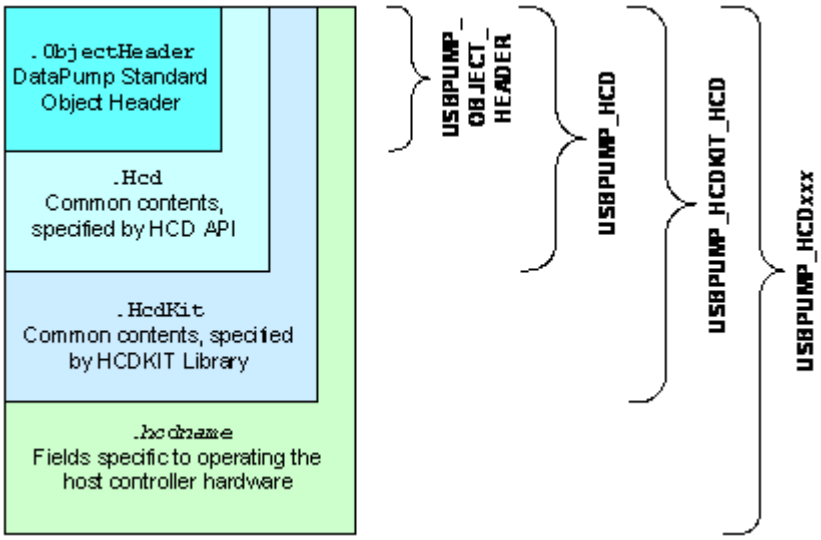
2. Access via the HCD `USBPUMP_HCD_INSWITCH`.

These two methods are fully under the control of the HCD implementation, and therefore we provide the possibility of writing a driver that implements all of its methods itself. However, that puts much responsibility for error checking and consistency into the drivers, rather than

allowing this common responsibility to be centralized. HCDs written in this way are called "*Full HCD Drivers*".

However, most HCD drivers are expected to be written using a common framework that provides most of the management operations in a centralized library. This library is called the "HCD kit", and such drivers are called "*HcdKit Drivers*". These drivers have an extended common HCD prefix, labeled "`.HcdKit`", as shown in Figure 8.

Figure 8. HcdKit HCD-Object Hierarchy



Provided full-driver HCD data structures are derived from the HcdKit hierarchy, full HCD drivers may also use portions of the HcdKit library.

9 HCD APIs

HCD APIs are of two kinds:

1. Data transfer APIs. Data transfer APIs are used by HCD clients
2. Control APIs. These APIs are primarily implemented using asynchronous IOCTls.

9.1 IOCTLs

HCD instances participate in the normal IOCTL discipline. In addition to the normal IOCTLs implemented by all objects, the following additional IOCTLs are passed through:

9.1.1 Common IOCTLs

Table 1 lists the common IOCTLs defined by the common HCD API.

Table 1. Common IOCTLs that are part of the HCD API

Name	Description
`USBPUMP_IOCTL_FUNCTION_OPEN`	Used by USBDI or the HCD test tool to open the HCD.
`USBPUMP_IOCTL_FUNCTION_CLOSE`	Used by USBDI or the HCD test tool to close the HCD.
`USBPUMP_IOCTL_FUNCTION_QUERY_PRESENT`	Poll whether the hardware associated with this function is really present. If not implemented by the HCD, clients shall assume that the HCD is permanently attached.

9.1.2 Platform IOCTLs

Platform IOCTLs are passed through to the UPLATFORM object.

9.1.3 OTG session control

See section 17.

9.1.4 Transceiver IOCTLs

See section 16.

9.1.5 HCD IOCTLs

Table 2 lists the common IOCTLs defined by the common HCD API.

Table 2 HCD IOCTLs that are part of the HCD API

Name	Description

`USBPUMP_IOCTL_HCD_GET_PORT_COUNT`
`Input: pHcd`
`Output: nRootHubs,nRootHubPorts`
	Obtains the total number of ports for the specified HCD.

`USBPUMP_IOCTL_HCD_CHECK_B_CONN`
`Input: pHcd, iPort`
`Output: fB_CONN_State`
	Obtain the B_CONN state of port for the specified HCD.

`USBPUMP_IOCTL_HCD_SET_PORT_FEATURE` `Input: pHcd, pRootHubDevice, iPort, fSet, Feature, Selector`	Set/Clear port feature

Name	Description
USBPUMP_IOCTL_HCD_GET_SCHEDULE_PARAMS Input: pHcd Output: nLevels, nTTLevels, LowSpeedHubClocks, LowSpeedHostDelay, FullSpeedHostDelay, HighSpeedHostDelay, MaxPeriodicClocks	Returns info about the HCD's scheduling capabilities.
USBPUMP_IOCTL_HCD_GET_ROOT_PORT_HANDOFF_FLAG Input: pHcd, portIndex, fReset Output: fState	Returns state of HCD's handoff flag.
USBPUMP_IOCTL_HCD_GET_OTG_CAPABLE Input: pHcdDevice, pPort, portIndex Output: fOtgCapable	Returns property of port that port is OTG capable or not
USBPUMP_IOCTL_HCD_GET_ADDRESS_BITMAP Input: pHcd Output: bAddressBitmap	Obtain the device address assignment bitmap for the specified HCD.

9.2 Submitting Transfer Requests

To submit a request, the client first allocates a USBPUMP_HCD_REQUEST packet, and then initializes it using the appropriate initialization API. Then the client submits the packet using the following API:

```
VOID (*pHcd->Hcd.InSwitch.pSubmitRequestFn)(
          USBPUMP_HCD *pHcd,
          USBPUMP_HCD_REQUEST *pHcdRequest,
          USBPUMP_HCD_REQUEST_DONE_FN *pDoneFn,
          VOID *pDoneInfo
          );
```

The request is processed asynchronously.

USBPUMP_HCD_REQUEST_DONE_FN has the following prototype:

```
typedef VOID
USBPUMP_HCD_REQUEST_DONE_FN(
          USBPUMP_HCD *pHcd,
          USBPUMP_HCD_REQUEST *pRequest,
          VOID *pDoneInfo,
          ARG_USTAT Status
          );
```

9.2.1 HcdKit Processing When Transfer Requests Are Submitted

HcdKit provides the request processing function `UsbPumpHcdKitI_SubmitRequest()`, which HcdKit will use in the InSwitch of HCDs supported by HcdKit. This function performs the following operations. Full HCD implementations must perform similar operations in order to guarantee compatibility with the DataPump USBDI.

- Verify that the input argument `pDoneFn` is non-`NULL`; otherwise just return.
- Verify that `pHcd` and `pRequest` are non-`NULL`; otherwise call `pDoneFn` directly and return.
- Make sure the length is at least the minimum length `(pRequest->Hdr.Length < sizeof(pRequest->Hdr)` or else call the done function from the input parameter list and give up without touching the request.
- Verify that `pRequest->Hdr.pHcd` points to the HCD, or fail the request without touching the request (apart from setting `Hdr.Status` to `USTAT_INVALID_PARAM`).
- Set `pRequest->Hdr.pDoneFn` and `pRequest->Hdr.pDoneInfo` from the input parameters.
- Clear `Hdr.InternalFlags`, `Hdr.pCancelFn`, `Hdr.pCancelInfo`, `Hdr.HcdReserved[0..n]`.
- Set `Hdr.Status` to `USTAT_BUSY`.
- If the hardware is down, complete the request immediately with appropriate status. Determine this from `pHcd->HcdKit.Status` bit `USBPUMP_HCD_STATUS_HW_UP`.
- Set `pRequest->Hdr.Refcount` to 1. (Now we can complete the request in the normal way)
- Verify the request code and length are compatible – if not, call the done function and give up. (This way, if the client submits a bogus request, the fault will happen in the client, not in HcdKit code.)
- Verify the pipe status pRequest->PipeControl.pPipe == NULL, return USTAT_STALL if pipe is halted.
- Validate the buffer for BUFFER_VALIDATE_NONE, BUFFER_VALIDATE_NULL, BUFFER_VALIDATE_FLAG.
- If `pHcd->HcdKit.HwSwitch.pVerifyRequestFn` is not `NULL`, call the verify function to further check the request. If the result is not `USTAT_OK`, complete the request by calling the done function directly.

- If `pRequest->Hdr.Timer == USBPUMP_HCD_TIMEOUT_DEFAULT`, and `pHcd->HcdKit.HwSwitch.pGetDefaultTimeout` is non-`NULL`, call it – it will adjust the timeout in place.

- If `pRequest->Hdr.TimeoutTimer.Ticks == USBPUMP_HCD_TIMEOUT_DEFAULT` (still), then set the timeout to `pHcd->HcdKit.DefaultTimeout`; this is required to be non-default.

- If `pRequest->Hdr.TimeoutTimer.Ticks == USBPUMP_HCD_TIMEOUT_NONE`, proceed to next step. Otherwise, increment `pRequest->Hdr.Refcount`, and set `pRequest->Hdr.Timer` to expire after the specified number of milliseconds. When it expires, arrange to call `UsbPumpHcdKitI_RequestTimeout(&pRequest->Hdr.Timer, timeval)`, which will set the timed-out flag, and will attempt to cancel the request. Also set the timer-active flag (`USBPUMP_HCD_REQUEST_INTERNAL_FLAG_TIMING`), so we know to cancel during the completion.

- Based on the request code, if this is a pipe-request, and if `pRequest->pPipe == pHcd->Hcd.RootHubDevice`, then sidetrack to root hub processing (section 9.2.2).

- Otherwise, if this is a pipe request, verify the pipe request code against the pipe type and direction. Fail if there's a mismatch.

- For Isoch transfers, validate the isoch descriptor against the buffer.

- For bulk transfers, validate the stream id.

- Finally call `pHcd->HcdKit.HwSwitch.pSubmitRequestFn(pHcd, pRequest)`, and return.

9.2.2 HcdKit Root-Hub Processing

If an operation targeting the default pipe of the root hub is received, HcdKit will process it as follows.

- If it's INIT_PIPE, DEINIT_PIPE or UPDATE_PIPE, complete it.

- If it's CONTROL_IN, and the setup is GET_DESCRIPTOR(HUB), decode (checking the fields that ought to be zero or fixed values) and pass the request to `pHcd->HcdKit.HwSwitch.pRootGetDescriptorFn()`.

- Otherwise, if it's CONTROL_IN and the setup is GET_HUB_STATUS, decode (checking the fields that ought to be zero or fixed values) and pass the request to `pHcd->HcdKit.HwSwitch.pRootGetHubStatusFn()`.

- CONTROL_IN and Setup is GET_STATUS [i.e., `bmRequestAttr` == 0xA0 and `bRequest` == GET_STATUS], decode (checking the fields that ought to be zero or fixed values) and pass to `UsbPumpHcdKitI_MapCopyToPioBuffer()`.

- CONTROL_IN and Setup is GET_PORT_STATUS [i.e., `bmRequestAttr` == 0xA3 and `bRequest` == GET_STATUS], decode (checking the fields that ought to be zero or fixed values) and pass to `pHcd->HcdKit.HwSwitch.pRootGetPortStatusFn()`.

- CONTROL_OUT and setup is SET_FEATURE(hub feature) [i.e., `bmRequestAttr` == 0x20 and `bRequest` == SET_FEATURE], decode (checking the fields that ought to be zero or fixed values) and pass to pHcd->HcdKit.HwSwitch. `pRootSetHubFeatureFn()` .

- CONTROL_OUT and setup is SET_FEATURE(port feature) [i.e., `bmRequestAttr` == 0x23 and `bRequest` == SET_FEATURE], decode (checking the fields that ought to be zero or fixed values) and pass to `UsbPumpHcdKitI_SetPortFeatureRequest()`. This includes handling port test status

- CONTROL_OUT and setup is CLEAR_FEATURE(hub feature) [i.e., `bmRequestAttr` == 0x20 and `bRequest` ==CLEAR_FEATURE], decode (checking the fields that ought to be zero or fixed values) and pass to `pHcd->HcdKit.HwSwitch.pRootClearHubFeatureFn()`.

- CONTROL_OUT and setup is CLEAR_FEATURE(port feature) [i.e., `bmRequestAttr` == 0x23 and `bRequest` == CLEAR_FEATURE], decode (checking the fields that ought to be zero or fixed values) and pass to `UsbPumpHcdKitI_ClearPortFeatureRequest()`. Unlike real hubs, we'll also accept requests to clear test mode features.

- If `pHcd->HcdKit.HwSwitch.pRootTtRequestFn` is not `NULL`, check whether this is any of the Root TT requests. If so, pass to `pHcd->HcdKit.HwSwitch.pRootTtRequestFn` for processing.

- INTERRUPT_IN: ignore the pipe type, and queue the request to the `(pHcd->HcdKit.HwSwitch.pRootHubSubmitStatusReadFn)()`. (Validate buffer and length first, though.)

- CONTROL_IN and setup is GET_DESCRIPTOR, decode (checking the fields that ought to be zero or fixed values) and pass to *pHcd->HcdKit.HwSwitch.pRootGetDescriptorFn().

- CONTROL_IN and setup is SET_AND_TEST, decode (checking the fields that ought to be zero or fixed values) and pass to *pHcd->HcdKit.HwSwitch.pRootLpmRequestFn().

- CONTROL_IN and setup is GET_PORT_ERR_COUNT, decode (checking the fields that ought to be zero or fixed values) and pass to pHcd->HcdKit.HwSwitch.pRootGetPortErrorCountFn().

- CONTROL_OUT and setup is SET_CONFIG, CLEAR_FEATURE, SET_HUB_DEPTH just complete the request by calling UsbPumpHcdRequest_Complete().

- CONTROL_OUT and setup is SET_DESCRIPTOR is unknown or not supported.

9.3 Canceling Transfer Requests

```
BOOL (*pHcd->Hcd.pInSwitch.pCancelRequestFn)(
        USBPUMP_HCD *pHcd,
        USBPUMP_HCD_REQUEST *pHcdRequest
        );
```

It is sometimes necessary to cancel a request. This API, which assumes that the request state is well defined, allows the caller to cancel a request.

Cancellation and timeout processing is considerably more complex within the HCD stack than it is in the DCD stack. This is because individual requests must be timed out and/or cancelled (especially on the default pipe for multi-function devices). The DCD stack in the DataPump avoids this complexity by disallowing cancellation of specific requests.

This API must only be called from within DataPump context, and HCD Requests can only be completed from within DataPump context. Therefore, before calling this request, the caller can easily determine whether the request is still "alive". However, there may be pending events in the event queue (behind this one) that will also attempt to reference the request directly. Further, the request may have a timeout pending. For this reason, each request has a reference count, which is managed internally by the HCD implementation as it processes the request. Only when the reference counter drops to zero will the client's completion routine be called.

10 HCD Data Structures

10.1 USBPUMP_HCD

USBPUMP_HCD is the generic object that represents a USB host controller.

A USBPUMP_HCD is the central context object for a given USB host controller instance within the DataPump.

This structure has the following fields

USBPUMP_OBJECT_HEADER ObjectHeader;

> The standard object header. The tag is |USBPUMP_HCD_TAG|. The IOCTL parent is the pointer to the next object closer to the |UPLATFORM| -- depending on the design of the HCD, there may be no objects, or several objects, in series between this HCD and the UPLATFORM. This may also be referenced as Hcd.ObjectHeader.

USBPUMP_HCD_CONTENTS Hcd;

The common definitions for all HCDs, defined as follows.

UPLATFORM *Hcd.pPlatform;

Pointer to the platform object, for convenience.

USBPUMP_HCD_INSWITCH Hcd.InSwitch;

The high speed APIs. This is inteded always to be supplied by the common HCD logic, but is separated out for API clarity and easy separation of functions.

USBPUMP_HCD_DEVICE **ppRootHubDevices;

The array of HCD device pointer. The device pointer for the root hub for this HCD. Generally speaking, pRootHubDevices[0] is pointer for the full speed root hub, pRootHubDevices[1] is for high speed root hub, and pRootHubDevices[2] is for super speed root hub.

10.1.1 Private Elements for use by the HCD

BYTES DefaultMaxTransferSize;

A configuration parameter, sets the max transfer size to be used by default for pipes on this HCD.

USBPUMP_HCD_REQUEST_QUEUE CompletionQueue;

A queue used by UsbPumpHcdRequest_Complete for deferring completion of requests.

UINT fEhciStartSplitOk: 1;

A flag indicating this HCD can handle a StartSplit and CompleteSplit in H-frame 1.

UINT fLateCompleteSplitsOk: 1;

A flag indicating that this HCD can handle Late Complete Splits. A Late Complete Split is a Complete Split that is in the mini-frame following the mini-frame the StartSplit is in.

10.2 USBPUMP_HCD_DEVICE

The USBPUMP_HCD_DEVICE object is used to model a device to the HCD. It contains information that's common to all devices that are connected to the HCD.

Part of the representation of a transfer target to the USBPUMP_HCD is the device address, transfer speed, and possibly (for full-speed devices on high-speed HCs) the hub address and hub port address of the TT on the high-speed bus.

Since this info is the same for every pipe on a device, we centralize it in the USBPUMP_HCD_DEVICE object.

This structure has the following fields.

UINT32 Tag;

The tag for this datastructure, 'UHdv' arranged in byte order so it dumps as "UHdv" in the debugger. This is filled in at init-time.

UINT32 uRouteString;

0:19 Route String as defined in USB 3.0

23 fHub, set if this device is a hub

24:31 bHubPortNumber, root hub port used to access this device

UINT8 bSpeed;

USBPUMP_DEVICE_SPEED_{LOW, FULL, HIGH}, forced into a UINT8. This is filled in at enum time for the device.

UINT8 bAddress;

The device address on its bus. This is filled in at enum time.

UINT8 bHubAddress;

For full-speed or high-speed only; this is the address of the device on its hub. This is filled in at enum time.

UNIT8 bHubPort;

For full-speed or high-speed only; this is the port of the device on its hub. This is filled in at enum time.

UINT8 bSlotID;

Slot ID as defined in XHCI.

UINT8 bTTHubSlotID;

For full-speed or high-speed hub only; this is the slot ID of the upstream high-speed hub.

UINT16 wMaxExitLatency;

The Maximum Exit Latency in milliseconds, indicates worst case time it takes to wake up all links in the path of this device.(USB 3.0)

USBPUMP_HCD_DEVICE_CAP Cap;

Device capabilities. It contains hub device capability, USB 2.0 extension capability, OTG device capability, wireless device capability and super speed device capability.

USBPUMP_HCD_PIPE DefaultPipe;

The default pipe for this device.

10.3 USBPUMP_HCD_PIPE

The `USBPUMP_HCD_PIPE` structure is used to model a pipe to the HCD.

HCDs use `USBPUMP_HCD_PIPE` instances to model logical pipes to their managed HC. These objects are allocated by USBD as part of the schedule creation process. The main purpose is to serve as a queue-head for the USBPUMP_HCD_REQUEST operations that target pipes.

The HCD client allocates memory for the `USBPUMP_HCD_PIPE` and initializes the fields. It then passes the `USBPUMP_HCD_PIPE` to the HCD using `USBPUMP_HCD_RQ_INIT_PIPE` (see 11.1). When the client is through using the pipe, it notifies the HCD using `USBPUMP_HCD_RQ_DEINIT_PIPE` (see 11.2); when the HCD completes the request, the client can recycle the memory holding the `USBPUMP_HCD_PIPE`. If the client needs to change the operating characteristics of the pipe, it waits for all pending requests on the pipe to complete, and then calls `USBPUMP_HCD_RQ_UPDATE_PIPE` (see 11.3).

This structure has the following fields.

```
UINT32 Tag;
```

The tag, 'UHpi' arranged in byte order so it dumps as "UHpi" in the debugger.

```
USBPUMP_HCD_DEVICE *pHcdDevice;
```

This points to the object that models the containing USB device to the HCD. (This allows for common storage of speed, address, default pipe, etc.) The USBPUMP_HCD_DEVICE in turn points to the host controller object USBPUMP_HCD.

```
UINT8 bEndpointAddress;
```

The endpoint address for this pipe (includes IN/OUT bit in bit 7).

```
UINT8 bEndpointType;
```

The endpoint type (bulk, interrupt, isoch, control)

`UINT16 wInterval;`

The polling interval for this pipe, in units of (micro)frames. If the pipe is for a full-speed or low-speed device, then this is in frames. If the pipe is for a high-speed device, then this is in microframes.

`UINT16 wMaxPacketSize;`

The maximum packet size for this endpoint, in a form that allows for easy comparison and computation. For high-bandwidth periodic endpoints, this is the base max-packet size (bits 10..0 of the endpoint descriptor). Bits 15..11 will always be zero. For other endpoints, this is the max-packet size.

`BYTES MaxTransferSize;`

The configured maximum transfer size for this pipe. This size is advisory to the HCD, and may be used by the HCD to optimize resource allocation.

`UINT8 bToggle;`

The data toggle to be used on the next transfer. Only valid when no transfer is pending. Initialized by client, updated by the HCD as transfer progresses. For isoch pipes, toggle is not used. Therefore, we reuse this as the count of the number of transactions to schedule per microframe.

If the client changes the data toggle independently of the HCD, the client must notify the HCD.

`UINT8 fHalted;`

If true, the pipe is halted. No transfers can be advanced from the trasnfer queue to the hardware. The hardware can set this byte; the client can set or reset thins.

`UINT8 StartSplitMask;`

This tells the HCD in which microframe to schedule the Start Split transactions.

`UINT8 CompleteSplitMask;`

This tells the HCD in which microframe to schedule the Complete Split transactions.

`UINT8 PeriodicLayer;`

For periodic endpoints only, this field gives the layer within the abstract binary schedule that this endpoint has been assigned to. Architecturally, this is required to be a number in the range from 0 to 7, corresponding to polling intervals of 1, 2, 4, ..., 128 (micro)frames. Frames are used for full/low speed pipes; microframes are used for high-speed pipes. This value will always be less than or equal to the binary tree depth reported by the HCD at initialization time.

If the client changes `PeriodicLayer` independently of the HCD, the client must notify the HCD.

`UINT8 PeriodicBucket;`

For periodic endpoints only, the field gives the offset within the abstract binary schedule tree for this endpoint. If many endpoints share the same periodic polling interval, USBDI will try to arrange to smooth out the traffic profile by starting in different frames/microframes. The HCD must arrange for the endpoint's traffic to be scheduled so that frame number modulo $2^{\texttt{PeriodicLayer}}$ is equal to `PeriodicBucket`. For low-speed and full-speed pipes, this is expressed in terms of frames. For high-speed pipes, this is expressed in terms of microframes.

`UINT8 nHighBandwidthPackets;`

For high-speed periodic endpoints only, this field gives the number of `wMaxPacketSize` packets to be transferred per frame. This will only be a value other than 1 if this is a high-bandwidth periodic endpoint.

UINT8 bMaxBurstSize;

USB 3.0. The maximum consecutive USB transactions that should be executed per scheduling opportunity. Valid values are from 1 to 16. SuperSpeed Endpoint Companion Descriptor:bMaxBurst + 1.

UINT16 wMaxStreamID;

USB 3.0: maximum stream ID supported. Only applicable to bulk endpoints. Values based on the endpoint companion descriptor are 0, 2, 4, 8, 16, .. 32,768, 65,533 (sic); but note that some HCs erroneously may not support (for example) a max StreamID of 32, but only a max ID of 31. So we store the max stream ID, which allows the HC to reduce the stream ID to something it can support.

A value of 0 indicates that the endpoint does not support streams.

UINT16 wBytesPerInterval;

USB 3.0. The total number of bytes this endpoint will transfer every service interval.

UINT8 bMult;

USB 3.0. A zero based value that determines the maximum number of packets within a service interval that this endpoint supports. From SuperSpeed Endpoint Companion Descriptor:bmAttributes Mult field. Set for periodic endpoints only. Always 0 for interrupt endpoint.

UINT8 fClearTT;

If true, the TT is in a state where it may not be able to pass traffic due to HCD cancellation, timeout or error. The HCD sets this flag and the client clears it after it clears the TT buffer in the hub.

10.3.1 Private Elements for use by the HCD

The following fields are provided by the portable header file, and are must not be used or referenced by the client while the USBPUMP_HCD_PIPE is owned by the HCD.

```
USBPUMP_HCD_REQUEST *pRequestQueue;
```

The pointer to the list of transfers pending for this pipe. Queued transfers are ultimately placed here. Items in this queue are not shared with the hardware, so aborts are easy.

```
USBPUMP_HCD_PIPE *pNext, *pLast;
```

Queue threads for managing pipes on a particular service list.

```
ADDRBITS_PTR_UNION HcdReserved[4];
```

Four entries that can be used as needed by the HCD for managing traffic on this pipe. (`ADDRBITS_PTR_UNION` is a union type, capable of holding a `UINT_PTR` integer or a `VOID` pointer.)

11 HCD Request Messages

This section defines each of the different `USBPUMP_HCD_REQUEST` messages, and indicates how clients should prepare them, and specifies how the HcdKit package handles them.

To allow for HCD-specific bookkeeping, `USBPUMP_HCD_PIPE` structures have fields reserved for use by the HCD.

11.1 USBPUMP_HCD_RQ_INIT_PIPE

Clients issue this request after allocating and initializing the client-owned fields of a `USBPUMP_HCD_PIPE`. Since the HCD implementation might be remote, this request is asynchronous. However, in many cases the initialization will complete immediately.

At the HCD level, this operation doesn't have anything to do with high-level bandwidth allocation. However, at the HCD level the HCD is obligated to assign the appropriate resources to allow subsequent requests to be issued to the pipe. In the case of periodic transfers, this means that the HCD will have do allocate any hardware-specific data structures that are needed for scheduling traffic for this pipe. Note that it's USBDI's responsibility to confirm that the

bandwidth is available prior to issuing the request. HCDs are not responsible for bandwidth management at this level.

It is not necessary to submit this high-level request to initialize the default pipe for the root hub.

```
VOID UsbPumpHcd_PreparePipeInitRequest(
          USBPUMP_HCD_REQUEST *pHcdRequest,
          USBPUMP_HCD_PIPE *pHcdPipe
          ARG_UINT8 Flags
          );
```

11.1.1 HcdKit Logic for USBPUMP_HCD_RQ_INIT_PIPE

HcdKit drivers must allocate any hardware or software resources needed to operate the pipe, or fail the request if sufficient resources are not available. For HCs that can handle arbitrary numbers of endpoints through system memory, memory must be allocated. For HCs that require time-multiplexing of pipes onto internal descriptors, the HCD may elect either only to support a limited number of pipes, or to implement time-multiplexing in software.

Remember that for periodic pipes, hardware-scheduling elements may need to be allocated (e.g., the interrupt descriptors that are used in the OHCI or EHCI).

Slots are available in the USBPUMP_HCD_PIPE structure for use by the HCDs in tracking allocated resources.

11.2 USBPUMP_HCD_RQ_DEINIT_PIPE

Clients issue this request when they are through using a given `USBPUMP_HCD_PIPE`. This call should be issued only when there are no requests pending on the pipe, before freeing the memory.

```
VOID UsbPumpHcd_PreparePipeDeInitRequest(
          USBPUMP_HCD_REQUEST *pHcdRequest,
          USBPUMP_HCD_PIPE *pHcdPipe
          ARG_UINT8 Flags
          );
```

HcdKit drivers must free any associated resources.

11.3 USBPUMP_HCD_RQ_UPDATE_PIPE

Clients issue this call when they have changed the state of the pipe. This allows hardware toggles, etc, to be updated, and also allows queue processing to restart if pipes have been halted due to errors.

```
VOID UsbPumpHcd_PreparePipeUpdateRequest(
          USBPUMP_HCD_REQUEST *pHcdRequest,
          USBPUMP_HCD_PIPE *pHcdPipe
          ARG_UINT8 Flags
          );
```

HcdKit drivers will only receive valid update requests from the HcdKit; they just need to copy data toggles to the hardware (if needed) and reset any halt conditions (resuming I/O), again if needed.

To change schedules or pipe types, first DEINIT and then re-INIT the pipe.

11.4 USBPUMP_HCD_RQ_BULK_IN, USBPUMP_HCD_RQ_BULK_OUT

These requests submit a bulk transfer to an HCD BULK pipe.

```
VOID UsbPumpHcd_PrepareBulkIntInRequest_V2(
          USBPUMP_HCD_REQUEST *pHcdRequest,
          USBPUMP_HCD_REQUEST_CODE RequestCode,
          USBPUMP_HCD_PIPE *pHcdPipe,
          UINT16 Timeout,
          ARG_UINT8 Flags,
          VOID *pBuffer,
          BYTES nBuffer,
          HANDLE A_hBuffer
          );

VOID UsbPumpHcd_PrepareBulkIntOutRequest_V2(
          USBPUMP_HCD_REQUEST *pHcdRequest,
          USBPUMP_HCD_REQUEST_CODE RequestCode,
          USBPUMP_HCD_PIPE *pHcdPipe,
          UINT16 Timeout,
          ARG_UINT8 Flags,
          CONST VOID *pBuffer,
          BYTES nBuffer,
          HANDLE A_hBuffer
          );
```

HcdKit drivers will only receive valid requests from the HcdKit, but are responsible for running interrupt pipes at or above the appropriate sampling rate.

11.5 USBPUMP_HCD_RQ_CONTROL_IN, USBPUMP_HCD_RQ_CONTROL_OUT

```
VOID UsbPumpHcd_PrepareControlInRequest_V2(
          USBPUMP_HCD_REQUEST *pHcdRequest,
          USBPUMP_HCD_PIPE *pHcdPipe,
```

```
          UINT16 HcdTimeout,
          ARG_UINT8 Flags,
          CONST USETUP_WIRE *pSetup,
          VOID *pBuffer,
          BYTES nBuffer,
          HANDLE A_hBuffer
          );

VOID UsbPumpHcd_PrepareControlOutRequest_V2(
          USBPUMP_HCD_REQUEST *pHcdRequest,
          USBPUMP_HCD_PIPE *pHcdPipe,
          UNIT16 HcdTimeout,
          ARG_UINT8 Flags,
          CONST USETUP_WIRE *pSetup,
          CONST VOID *pBuffer OPTIONAL,
          BYTES nBuffer,
          HANDLE A_hBuffer
          );
```

Prepare a control request.

HcdKit drivers will not receive control requests targeting the root hub. (In fact, none of the root hub operations will go through the normal path; instead they'll be decoded and dispatched to the appropriate root-hub entry points.)

According to the control transfer protocol outlined in section 8.5.3 of [USBCORE], there are three different patterns of transactions for a control transfer. The pattern used is determined by the direction of data transfer, and the number of bytes to be transferred during the data phase. These patterns are:

- For control-out transfers, the pattern is SETUP, OUT +, IN(ZLP).
- For control-in transfers, the pattern is SETUP, IN+, OUT(ZLP).
- For control transfers with no data-phase, the pattern is SETUP, IN(ZLP).

HCD Clients are required to send control transfers with no data phase using `USBPUMP_HCD_RQ_CONTROL_OUT`, with a zero-length transfer and a NULL transfer buffer pointer.

11.6 USBPUMP_HCD_RQ_ISOCH_IN, USBPUMP_HCD_RQ_ISOCH_OUT

```
VOID UsbPumpHcd_PrepareIsochInRequest_V2(
          USBPUMP_HCD_REQUEST *pHcdRequest,
          USBPUMP_HCD_PIPE *pHcdPipe,
          UINT16 HcdTimeout,
          ARG_UINT8 Flags,
          UINT32 IsochStartFrame,
```

```
        USBPUMP_ISOCH_PACKET_DESCR *pIsochDescr,
        BYTES sizeIsochDescrs,
        VOID *pBuffer,
        BYTES nBuffer,
        HANDLE A_hBuffer
        );

VOID UsbPumpHcd_PrepareIsochOutRequest_V2(
        USBPUMP_HCD_REQUEST *pHcdRequest,
        USBPUMP_HCD_PIPE *pHcdPipe
        UINT16 HcdTimeout,
        ARG_UINT8 Flags,
        UINT32 IsochStartFrame,
        USBPUMP_ISOCH_PACKET_DESCR*pIsochDescr,
        BYTES sizeIsochDescrs,
        CONST VOID *pBuffer,
        BYTES nBuffer,
        HANDLE A_hBuffer
        );
```

Set up an isoch transfer. `pBuffer` is composed of a series of packets. Unlike for DCDs, the packets are identified using an external vector of `USBPUMP_HCD_ISOCH_DESCR` nodes. Each node specifies the location of a single packet within the buffer; the node also has a slot for status and the size of the packet. The status and size are updated as each packet is transferred.

HcdKit drivers will only receive requests that have been validated.

11.7 USBPUMP_HCD_RQ_GET_FRAME

This request gets the current frame, along with information about how to convert the frame number into milliseconds.

```
VOID UsbPumpHcd_PrepareGetFrameRequest(
        USBPUMP_HCD_REQUEST *pHcdRequest,
        UINT16 HcdTimeout,
        ARG_UINT8 Flags
        );
```

The data is returned in the request. pHcdRequest->GetFrame is of type USBPUMP_HCD_REQUEST_GET_FRAME, and has the following request-specific fields.

`UINT32 StandardFrame;` — The "standard" frame count, in milliseconds. This value is always compatible with full-speed USB. The frame count is returned as a 32-bit number and must be maintained by the HCD as a 32-bit number.

`UINT64 NativeFrame;` — The "bus native" frame count. The units of this number are not necessarily in milliseconds. To convert to milliseconds exactly, use the numerator and denominator fields, by multiplying by the numerator,

and dividing by the denominator.

`UINT32 Numerator;` — The numerator for converting native frame count to milliseconds (multiply by this before dividing by denominator).

`UINT32 Denominator;` — The denominator for converting native frame count to milliseconds. Divide by this after multiplying by the denominator.

Some values of numerator and denominator: for a full speed bus, Numerator will be 1, and Denominator will also be 1. For a high-speed bus, Numerator will be 1, and Denominator will be 8. For a Wireless USB bus, Numerator will be 125 and Denominator will be 128, which is the simplified form of the formal ratio of 1000/1024.

12 HcdKit Structures

12.1 HCDKIT HCDs: `USBPUMP_HCDKIT_HCD`

USBPUMP_HCDKIT_HUBPORT_STATUS

This structure represents the status and change indications for a single hub or port. USB Hubs and ports share a common status reporting structure: a 16=bit status word, followed by a 16 bit change word.

This structure has the following fields

UINT16 wStatus;

> This contains a status mask, in native byte order. The bits are as defined in "usbhub20.h", USB_Hub_HUB_STATUS_wStatus_... or USB_Hub_PORT_STATUS_wStatus_... as appropriate.

UINT16 wChange;

> This contains a change mask. Same bit definitions as for wStatus, but a set bit indicates that a status change has occurred. Changes are reset using CLEAR_FEATURE.

USBPUMP_HCDKIT_ROOTHUBPORT

This structure represents the root hub port information. HcdKit virtual root hub logic uses this structure to control root hub port.

This structure has the following fields

USBPUMP_HCDKIT_ROOTHUB *pRootHub;

> The pointer for the root hub.

USBPUMP_HCDKIT_HUBPORT_STATUS PortStatus;

Root hub port status.

UINT8 PortNumber;

Port number for this root hub port.

UINT8 bSpeed;

Current port speed. It is same as connected device speed.

UINT8 State;

Port state for this root hub port. This will be used by HCD to maintain state of the root hub port.

UINT8 fRunTimer;

TRUE if timer in running.

ADDRBITS_PTR_UNION HcdReserved;

Reserved for the HCD usage.

USBPUMP_HCDKIT_ROOTHUB

This structure represents the root hub device information. HcdKit virtual root hub logic uses this structure to control root hub device

This structure has the following fields

USBPUMP_HCD_DEVICE RootHubDevice;

The device representation (and the default pipe) for the root hub for this HCD.

USBPUMP_HCD_PIPE RootHubStatusPipe;

The interrupt pipe for the root hub for this HCD.

USBPUMP_HCDKIT_HUBPORT_STATUS RootHubStatus;

The root hub status, expressed in the same form returned by the get-hub-status operation: one word of status, one word of change. Note, however, that the words are stored in local native order.

UINT8 *pRootPortStatusChange;

Base of array of port status change.

USBPUMP_HCD_REQUEST *pRootHubRequest;

The pointer of HCD request to get notification from root hub.

USBPUMP_HCDKIT_ROOTHUBPORT *pRootHubPorts;

Base of array of root hub port entries, indexed by root port number.

USBPUMP_HCDKIT_HCD_CONTENTS

This structure represents an HcdKit based HCD instance object to HcdKit modules. HcdKit common logic uses objects of type USBPUMP_HCDKIT_HCD to represent HCD instances. These objects are unions, which combine a hierarchy of views. In addition, the objects are always a proper subset of the total data structure; the terminating layer of the HCD implementation normaly requires additional data of its own.

This structure has the following fields

USBPUMP_HCD HcdKit.HcdCast;

A view of the object that is type-compatible with functions that require USBPUMP_HCD pointers. This allows you to make the conversion without doing a cast.

USBPUMP_OBJECT_HEADER ObjectHeader.*;

The fields defined in the standard object header.

USBPUMP_HCD_CONTENTS Hcd;

The fields defined in the standard HCD.

UINT32 HcdKit.OpenCount;

Function open count for use by HcdKit.

USBPUMP_HCDKIT_SWITCH HcdKit.HwSwitch;

The methods provided by the hardware-specific code for use by HcdKit.

UINT32 HcdKit.Status;

Hardware status for use by HcdKit. This is a bit-map.

If USBPUMP_HCDKIT_STATUS_HW_UP is set, then the hardware is operational, and requests should be passed to the next hardware level for processing. If reset, then the hardware is not operational, and all normal USB requests into the HCD from above will be rejected without being passed to the hardware-specific code.

USBPUMP_TIMER_TIMEOUT HcdKit.DefaultTimeout

Default HCD request timeout for use by HcdKit.

BYTES nRootPorts;

Base of array of status entries, indexed by root port number. nRootPorts gives the number of root ports.

USBPUMP_HCDKIT_ROOTHUB *HcdKit.pRootHubs;

The array of root hub device pointer. The root hub device pointer for the root hub device for this HCD.

12.1.1 HCD status (`HcdKit.Status`)

Table 3. Bits in `HcdKit.Status`

Name	Description
USBPUMP_HCDKIT_HCD_STATUS_HW_UP	This flag is controlled by the HCD-specific code, and should be set when the root port can accept root hub requests. Presumably, the root hub status will indicate whether access to subordinate layers is appropriate. Normally (for non-removable hardware) this is set during initialization. QQQ when this changes, we should send a notification to USBD. Specify this notification, implement it in USBD

13 HcdKit Hardware API functions

13.1 IOCTL Processing

The IOCTL processing function:

```
USBPUMP_IOCTL_RESULT (*pHcd->HcdKit.HwSwitch.pIoctlFn)(
          USBPUMP_HCDKIT_HCD *pHcd,
          USBPUMP_IOCTL_CODE Code,
          CONST VOID *pInArg,
          VOID *pOutArg
          );
```

Provides hardware-specific IOCTL processing.

13.2 Validate Request

This function, if not `NULL`, will be called by HcdKit during early request processing to validate the request. This allows the HCD to perform additional validation or else to reject unsupported commands early.

```
USTAT (*pHcd->HcdKit.HwSwitch.pValidateRequestFn)(
          USBPUMP_HCDKIT_HCD *pHcd,
          USBPUMP_HCD_REQUEST *pRequest
          );
```

Return `USTAT_OK` if the request is OK, otherwise some error code to be used in completing the request.

13.3 Submit Request

```
VOID (*pHCD->HcdKit.HwSwitch.pSubmitRequestFn)(
          USBPUMP_HCDKIT_HCD *pHcd,
          USBPUMP_HCD_REQUEST *pRequest
          );
```

13.4 Get Root Hub Descriptor

```
VOID (*pHcd->HcdKit.HwSwitch.pRootGetDescriptorFn)(
        USBPUMP_HCDKIT_HCD *pHcd,
        USBPUMP_HCDKIT_ROOTHUB * pRootHub,
        USBPUMP_HCD_REQUEST *pRequest,
        VOID *pBuffer,
        BYTES nBytes
        );
```

Reads the root hub descriptor into the buffer, and completes the request, possibly asynchronously.

13.5 Set Root Hub Feature

```
VOID (*pHcd->HcdKit.HwSwitch.pRootSetHubFeatureFn)(
          USBPUMP_HCDKIT_HCD *pHcd,
          USBPUMP_HCDKIT_ROOTHUB *pRootHub,
          USBPUMP_HCD_REQUEST *pRequest,
          ARG_UINT16 Feature
          );
```

Set the specified root hub feature and complete the request, possibly asynchronously.

13.6 Clear Root Hub Feature

```
VOID (*pHcd->HcdKit.HwSwitch.pRootClearHubFeatureFn)(
          USBPUMP_HCDKIT_HCD *pHcd,
          USBPUMP_HCDKIT_ROOTHUB *pRootHub,
          USBPUMP_HCD_REQUEST *pRequest,
          ARG_UINT16 Feature
          );
```

Clear the specified root hub feature and complete the request, possibly asynchronously.

13.7 Set Root Port Feature

```
VOID (*pHcd->HcdKit.HwSwitch.pRootSetPortFeatureFn)(
          USBPUMP_HCDKIT_HCD *pHcd,
          USBPUMP_HCDKIT_ROOTHUB *pRootHub,
          USBPUMP_HCD_REQUEST *pRequest,
          ARG_UINT16 Feature,
          ARG_UINT8 Port,
          ARG_UINT8 Selector
          );
```

Set the specified root port feature and complete the request, possibly asynchronously.

13.8 Clear Root Port Feature

```
VOID (*pHcd->HcdKit.HwSwitch.pRootClearPortFeatureFn)(
          USBPUMP_HCDKIT_HCD *pHcd,
          USBPUMP_HCDKIT_ROOTHUB *pRootHub,
          USBPUMP_HCD_REQUEST *pRequest,
          ARG_UINT16 Feature,
          ARG_UINT8 Port,
          ARG_UINT8 Selector
          );
```

Clear the specified root port feature and complete the request, possibly asynchronously.

13.9 Get Hub Status

```
VOID (*pHcd->HcdKit.HwSwitch.pRootGetHubStatusFn)(
          USBPUMP_HCDKIT_HCD *pHcd,
          USBPUMP_HCDKIT_ROOTHUB *pRootHub,
          USBPUMP_HCD_REQUEST *pRequest,
          VOID *pBuffer
          );
```

The root port status is placed in the 4-byte buffer at `*pBuffer`. HcdKit will validate the request before passing it down, so there's no need for the buffer size.

13.10 Get Port Status

```
VOID (*pHcd->HcdKit.HwSwitch.pRootGetPortStatusFn)(
          USBPUMP_HCDKIT_HCD *pHcd,
          USBPUMP_HCDKIT_ROOTHUB *pRootHub,
          USBPUMP_HCD_REQUEST *pRequest,
          VOID *pBuffer,
          ARG_UINT8 Port
          );
```

The root port status is placed in the 4-byte buffer at `*pBuffer`. HcdKit will validate the request before passing it down, so there's no need for the buffer size

13.11 Submit Root Hub Status Read

```
VOID (*pHcd->HcdKit.HwSwitch.pRootSubmitStatusReadFn)(
          USBPUMP_HCDKIT_HCD *pHcd,
          USBPUMP_HCDKIT_ROOTHUB *pRootHub,
          USBPUMP_HCD_REQUEST *pRequest
          );
```

This function is used to submit a status-read to the root hub. The HCD must queue this until a status change is detected (using a polling loop, if needed). Data should be returned as per [USB2.0] section 11.12.4. This is normally completed asynchronously; status should be polled periodically much as interrupt pipes are.

13.12 Process Root Hub TT Operations

Some root hubs on high-speed devices may have embedded TTs. In that case, the HCD must provide the following method:

```
VOID (*pHcd->HcdKit.HwSwitch.pRootTtRequestFn)(
          USBPUMP_HCDKIT_HCD *pHcd,
          USBPUMP_HCDKIT_ROOTHUB *pRootHub,
          USBPUMP_HCD_REQUEST *pRequest,
          ARG_UINT8 Port
          );
```

The HCD is responsible for decoding the request, which is a valid root-hub TT request.

14 Hcd Support Routines

The In Switch normally contains the following routines, which are provided by the common library.

14.1.1 Setting a Cancel Routine -- `UsbPumpHcdRequest_SetCancelRoutine`

Setting a non-`NULL` cancel routine also increments the reference count.

```
USBPUMP_HCD_REQUEST_CANCEL_FN UsbPumpHcdRequest_SetCancelRoutine(
          USBPUMP_HCD_REQUEST *pRequest,
          USBPUMP_HCD_REQUEST_CANCEL_FN *pCancelFn,
          VOID *pCancelInfo
          );
```

This routine always returns the previous cancel function pointer.

If the cancel function was `NULL` and is now non-`NULL`, the reference count of `pRequest` is incremented.

If the cancel function was non-`NULL` and is now non-`NULL` then the reference count is left alone.

If the cancel function was non-`NULL`, and is now `NULL`, then the reference counter is decremented.

If the cancel function was `NULL` and is now `NULL`, the reference counter is not changed.

The cancel info in the request is only updated if `pCancelFn` is not `NULL`.

14.1.2 Completing an HCD Request (`UsbPumpHcdRequest_Complete`)

To complete an HCD request, HCD functions call:

```
BOOL UsbPumpHcdRequest_Complete(
          USBPUMP_HCD_REQUEST *pRequest,
          USBPUMP_HCD_REQUEST **ppQueueHead,
          USTAT Status
          );
```

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

Completion is complicated. If a timeout is pending, we cancel the timer. We first take the request of the specified queue, updating the queue head. Then we reset the cancel routine (and decrement the use count if it was previously set). Then, if a timeout is pending, we cancel the timer (decrementing the ref count if we succeed in canceling it).

Finally, if the ref count is now 1, we post the request into the HCD's request completion queue. This arranges for the DataPump event dispatcher to call the user's done function to complete the request.

Otherwise either the timer still holds a reference or some other software module holds a reference. We can 't complete until the ref count goes to zero, but the other function (by holding a ref count) guarantees that a completion will happen, probably soon.

15 Queues of HCD Requests

The `USBPUMP_HCD_REQUEST_QUEUE` object is used for serializing and processing HCD requests.

To initialize a queue, call:

```
VOID UsbPumpHcdRequestQueue_Initialize_V1(
          USBPUMP_HCD_REQUEST_QUEUE *pQueue,
          USBPUMP_HCD_REQUEST_QUEUE_ARRIVAL_FN *pArrivalFn,
          VOID *pContext,
          UPLATFORM *pPlatform
          );
```

The request queue arrival function shall have the following prototype:

```
typedef VOID USBPUMP_HCD_REQUEST_QUEUE_ARRIVAL_FN(
          USBUMP_HCD_REQUEST_QUEUE *pQueue,
          VOID *pContext
          );
```

It will be called each time the queue changes state from empty to non-empty.

To put a queue element into a queue, use:

```
VOID UsbPumpHcdRequestQueue_PutRequest(
          USBPUMP_HCD_REQUEST_QUEUE *pQueue,
          USBPUMP_HCD_REQUEST *pRequest
          );
```

16 USBPHY API

The USBPHY API is provided using DataPump IOCTL operations. There are two classes of IOCTLs.

1. Inward IOCTLs which provide services for clients that are further from the hardware

2. Outward notifications, to alert registered clients that something has changed at the OTG interface.

The requirements for this API are:

USBPHY-001 Clients shall be able to detect the capabilities of the transceiver.

USBPHY-002 The API shall be asynchronous where necessary in order to support I2C transceivers or transceivers that are on remote USB busses (WUSB DWAs, etc).

16.1 USBPHY IOCTL operations

The following IOCTLs are defined in `phy/i/usbioctl_phy.h`:

Table 4. USBPHY IOCTL Operations

Operation	Description
USBPUMP_IOCTL_USBPHY_GET_GLOBAL_CAPABILITIES_ASYNC Output: NumPorts, HubCcharacteristics, PowerOnToPowerGood, HubControlCurrent, RemovablePorts, OtgPorts, DevOnlyPort, SpecialPorts	Returns a capabilities description structure for the entire transceiver (might be multi port)
USBPUMP_IOCTL_USBPHY_GET_PORT_CAPABILITIES_ASYNC Output: PortMaxPower	For a given port, return the port's capabilities. These include: Device operation possible? Host operation possible? Speed support bitmap (low/full/high x host/device)
USBPUMP_IOCTL_USBPHY_GLOBAL_POWER_ENABLE_ASYNC Input: fEnable	Power transceiver up or down – fEnable specifies the desired state. If TRUE, and the transceiver is disabled, the transceiver should move to the IDLE state and power up. If transceiver is already enabled, this is a no-op. If FALSE, and transceiver is enabled, this should force the transceiver into disabled state, and transceiver should send appropriate messages (END_SESSION, DISCONNECT) if in the active state. If transceiver already disabled, this is a no-op. If the transceiver is an OTG transceiver, this

Operation	Description
	function should not be used directly, instead use the OTG IOCTL_OTGCD_A_BUS_REQUEST to turn power on, and IOCTL_OTGCD_A_BUS_DROP to turn power off.
USBPUMP_IOCTL_USBPHY_PORT_POWER_ENABLE_ASYNC Input: iPort, fEnable	Turn on power on a given port; the port must be a host port, and must be in HOST or IDLE state.
USBPUMP_IOCTL_USBPHY_GET_GLOBAL_POWER_ASYNC Output: fPower(phy), wStatus(hub), wChange(hub)	Receive the power state for the phy itself, and root hub status if appropriate.
USBPUMP_IOCTL_USBPHY_GET_PORT_STATUS_ASYNC Input: iPort Output: PhyState, wStatus, wChange, DeviceFlags	Receive the status for a given port
USBPUMP_IOCTL_USBPHY_REGISTER_HCD Input: pHcd, pHcdEventFn, pContext, pHcdIsrFn	Register an HCD and establish a notification callback function.
USBPUMP_IOCTL_USBPHY_REGISTER_DCD Input: pDcd, pDcdEventFn, pContext, pDcdIsrFn	Register a DCD and establish a notification callback function.
USBPUMP_IOCTL_USBPHY_RELEASE_PORT_ASYNC Input: iPort	Put port back in IDLE state, able to accept either a host connect or a device connect.
USBPUMP_IOCTL_USBPHY_SUSPEND_PORT_ASYNC Input: iPort, fSuspend	Only if in a host state: put the port in a suspend mode.
USBPUMP_IOCTL_USBPHY_CONNECT_USB_ISR Input: pUsbIsr, pUsbIsrCtx, pOldUsbIsr, pOldUsbIsrCtx	Connect PHY interrupt service routine
USBPUMP_IOCTL_USBPHY_CLEAR_B_CONN	Clear **b_conn** variable in the OTG state machine
USBPUMP_IOCTL_USBPHY_CHECK_B_CONN_ASYNC Output: fB_CONN_State	Check state of **b_conn** variable in the OTG state machine
USBPUMP_IOCTL_USBPHY_CHECK_POWER Input: Port, maxCurrent	Check whether a port can supply the specified amount of power (optional)

Operation	Description
Output: Status	
USBPUMP_IOCTL_USBPHY_BOOK_POWER Input: Port, maxCurrent Output: Status	Record that a client has requested that the specified amount of power be allocated to the specified port of the phy (optionally implemented).
USBPUMP_IOCTL_USBPHY_FREE_POWER Input: Port, maxCurrent	Release a request for power (optionally implemented).

16.2 HCD Callback Functions

16.2.1 USBPUMP_USBPHY_HCD_EVENT_FN

The HCD callback function has the following prototype:

```
VOID USBPUMP_USBPHY_HCD_EVENT_FN(
          USBPUMP_OBJECT_HEADER *pHcd,
          VOID *pCallBackInfo,
          USBPUMP_USBPHY_HCD_EVENT eventCode,
          VOID *pEventSpecificInfo
          );
```

The HCD-specific events are:

`USBPUMP_USBPHY_HCD_EVENT_GOT_DEVICE`	Sent when a device is detected. The HCD is responsible for querying the speed from its internal registers. `pEventSpecificInfo` is a pointer to a `USBPUMP_USBPHY_EVENT_GOT_DEVICE_INFO` object., giving the device speed.
`USBPUMP_USBPHY_HCD_EVENT_CABLE_DETACH`	Sent when transceiver detects a cable detach. For OTG mini A-B connectors, this will detect a device disconnect from the local socket, but will not detect a device disconnect from the far end of the cable. If the far end is detached, then the HCD is responsible for detecting the fact that the idle state of the bus is SE0 rather than J. (J is D+ == 1, D- == 0 for full speed, D+ == 0, D- == 1 for low speed). HCD should send `USBPUMP_IOCTL_USBPHY_RELEASE_PORT` to indicate that it has finished processing. This is passed a pointer to a `USBPUMP_USBPHY_HCD_EVENT_CABLE_DETACH_INFO` structure, giving the port number.
`USBPUMP_USBPHY_HCD_EVENT_STATE_CHANGE`	Sent when HCD state has changed. `pEventSpecificInfo` is a pointer to a

	USBPUMP_USBPHY_EVENT_HCD_STATE_CHANGE_INFO object. Note that presently the contents of the info structure are not defined.
USBPUMP_USBPHY_HCD_EVENT_POWER_INVALID	Sent when power invalid.
USBPUMP_USBPHY_HCD_EVENT_REMOTE_WAKEUP	Sent when remote wakeup.
USBPUMP_USBPHY_HCD_EVENT_PORT_IDLE	Sent when HCD port is idle.

16.2.2 USBPUMP_USBPHY_EVENT_GOT_DEVICE_INFO

```
typedef struct
         {
         UINT32                Port;  // port to which it's attached.
         USBPUMP_DEVICE_SPEED Speed;  // speed of device
         } USBPUMP_USBPHY_EVENT_GOT_DEVICE_INFO;
```

16.2.3 USBPUMP_USBPHY_HCD_EVENT_CABLE_DETACH_INFO

```
typedef struct
         {
         UINT32              Port;   // port to which it's attached.
         } USBPUMP_USBPHY_HCD_EVENT_CABLE_DETACH_INFO;
```

16.2.4 USBPUMP_USBPHY_HCD_EVENT_STATE_CHANGE_INFO

```
typedef struct
         {
         UINT32                  Port;   // port to which it's attached.
         USBPUMP_USBPHY_STATE    OldState;
         USBPUMP_USBPHY_STATE    NewState;
         } USBPUMP_USBPHY_HCD_EVENT_STATE_CHANGE_INFO;
```

16.2.5 USBPUMP_USBPHY_HCD_EVENT_POWER_INVALID

```
typedef struct
         {
         UINT32                  Port;   // port to which it's attached.
         } USBPUMP_USBPHY_HCD_EVENT_POWER_INVALID_INFO;
```

16.2.6 USBPUMP_USBPHY_HCD_EVENT_REMOTE_WAKEUP

```
typedef struct
```

```
        {
        UINT32                  Port;   // port to which it's attached.
        } USBPUMP_USBPHY_HCD_EVENT_REMOTE_WAKEUP_INFO;
```

16.2.7 USBPUMP_USBPHY_HCD_EVENT_PORT_IDLE

```
typedef struct
        {
        UINT32                  Port;   // port to which it's attached.
        BOOL                    fIdle;
        } USBPUMP_USBPHY_HCD_EVENT_PORT_IDLE_INFO;
```

16.3 DCD Callback Functions (USBPUMP_USBPHY_DCD_EVENT_FN)

The DCD Callback function has the following prototype.

```
VOID USBPUMP_USBPHY_DCD_EVENT_FN(
        USBPUMP_OBJECT_HEADER *pDcd,
        VOID *pCallBackInfo,
        USBPUMP_USBPHY_DCD_EVENT eventCode,
        VOID *pEventSpecificInfo
        );
```

The DCD-specific events are:

USBPUMP_USBPHY_DCD_EVENT_START_SESSION	Sent when Vbus is detected. pEventSpecificInfo points to a USBPUMP_USBPHY_DCD_EVENT_START_SESSION_INFO structure which gives the port number (usually not needed).
USBPUMP_USBPHY_DCD_EVENT_END_SESSION	Sent when Vbus becomes invalid. DCD should send USBPUMP_IOCTL_USBPHY_RELEASE_PORT to indicate that it has finished processing. pEventSpecificInfo is NULL.
USBPUMP_USBPHY_DCD_EVENT_STATE_CHANGE	Sent when DCD state has changed.

17 OTGCD API

The OTGCD API is provided using DataPump IOCTL operations. There are potentially two classes of IOCTLs that are implemented:

1. Inward IOCTLs which provide services for clients that are further from the hardware.

2. Outward notifications, to alert registered clients that something has changed at the OTG interface.

The requirements for this API are:

OTGCD-001 Clients shall be able to discover the presence/absence of OTG capabilities.

OTGCD-002 The HCD and DCD shall both be able to be clients of the OTGCD layer.

OTGCD-003 HCD and DCD shall require no changes when used without OTG support and with limited functionality OTG transceivers

OTGCD-004 The API shall be asynchronous where necessary in order to support I2C transceivers or transceivers that are on remote USB busses (WUSB DWAs, etc).

17.1 OTG IOCTLs

The following IOCTLs are defined:

Table 5. USBPHY IOCTL Requests used by OTGCD.

IOCTL Request	Description
USBPUMP_IOCTL_USBPHY_GET_GLOBAL_CAPABILITIES_ASYNC Output: NumPorts, HubCharacteristics, PowerOnToPowerGood, HubControlCurrent, RemovablePorts, OtgPorts, DevOnlyPort, SpecialPorts	Returns a capabilities description structure.
USBPUMP_IOCTL_USBPHY_REGISTER_HCD Input: pHcd, pHcdEventFn, pContext, pHcdIsrFn	Connects a `USBPUMP_HCD` to the OTG transceiver instance, and registers a notification function.
USBPUMP_IOCTL_USBPHY_REGISTER_DCD Input: pDcd, pDcdEventFn, pContext, pDcdIsrFn	Register a DCD and establish a notification callback function. Connects a `UDEVICE` to the OTG transceiver instance. The OTG transceiver should register for UEVENTS, if needed. NB: if the transceiver support is integrated into the silicon, as on the ISP1362, this request might not be implemented or might be done directly as part of the DCD initialization.
USBPUMP_IOCTL_USBPHY_CHECK_POWER Input: Port, maxCurrent Output: Status	Check whether a port can supply the specified amount of power
USBPUMP_IOCTL_USBPHY_BOOK_POWER Input: Port, maxCurrent	Record that a client has requested that the specified amount of power be

IOCTL Request	Description
Output: Status	allocated to the specified port of the phy.
USBPUMP_IOCTL_USBPHY_FREE_POWER Input: Port, maxCurrent	Release a request for power.

Table 6. OTGCD Additional IOCTL Requests

IOCTL Request	Description
`USBPUMP_IOCTL_OTGCD_ENABLE_HNP (pArg,fEnable)`	Tells the transceiver (which must be in host mode) to prepare for HNP when the HCD releases the port. Essentially the same as setting **`a_set_b_hnp_en`**.
`USBPUMP_IOCTL_OTGCD_A_SUSPEND_REQ (pArg, fState)`	Set (or clear) **`a_suspend_req`** variable to request that the host bus be suspended.
`USBPUMP_IOCTL_OTGCD_A_BUS_REQUEST (pArg, fEnable)`	Tells the OTG state machine that the upper layers are requesting a bus session. This directly kicks the finite state machine.
`USBPUMP_IOCTL_OTGCD_A_BUS_DROP (pArg, fDrop)`	Tells the OTG state machine that the upper layers have finished a bus session. If this is TRUE, then bus request is ignored.
`USBPUMP_IOCTL_OTGCD_A_CLR_ERR`	Tell the OTG state machine to clear any current error. (The FSM will automatically reset this)
`USBPUMP_IOCTL_OTGCD_B_BUS_REQUEST (pArg, fEnable)`	Tells the OTG state machine that the system software wants to run a B-bus session. This will cause SRP, etc., as needed. This will automatically reset on any transition out of the B state.
`USBPUMP_IOCTL_OTGCD_SET_B_CONNECT_TIMER (pArg, ulMilliseconds)`	Set the B-connect timer value to n milliseconds. The minimum value is 1000 milliseconds (1 second) (per table 5-2 of the specification); values of 0 or greater than the range of the timer are interpreted as “indefinite” (i.e., the port will remain in **`a_wait_bcon`** indefinitely); values between 1 and 999 are treated as errors. Setting this value while the system is waiting for a B connection doesn’t affect the timing of the current connection attempt.
`USBPUMP_IOCTL_OTGCD_GET_STATE_ASYNC (OtgVars, ulConnectTimer)`	Gets a snapshot of the OTG FSM’s state. Asynchronous because otherwise there’s no way to put the FSM remotely (the other IOCTLs above could be forwarded asynch to another process for completion in the

IOCTL Request	Description
	background). Returns a USBPUP_IOCTL_OTGCD_GET_STATE_ASYNC_ARG filled in with the OTG state vector.
`USBPUMP_IOCTL_OTGCD_ENABLE_SRP_BASED (pArg,fEnable)`	Tells the OTG state machine that the system software want to set SRP based or insertion based mode of OTG device when in an A-Role. Insertion based mode is default mode.

In order to support the OTG descriptor and functionality, a core IOCTL for HNP has been added to the DataPump. This IOCTL should be implemented and claimed by the OTG transceiver module.

Table 7. Core IOCTLs for OTG

IOCTL	Description
`USBPUMP_IOCTL_DCD_SESSION_REQUEST`	Tells the OTG state machine that the system software wants to run a B-bus session, but doesn't want to be host. This will cause SRP, etc., as needed. This is a one-shot; the FSM will automatically reset this after doing SRP. The core device management code will issue this if it gets a remote-wakeup request, but finds that power is off. Of course, if this is not an OTG system, this probably won't be claimed.
`USBPUMP_IOCTL_DCD_HNP_ENABLED (fState)`	If `fState` is true, tell the OTG state machine that the DCD has received a `SET_FEATURE (b_hnp_enable)` request from the host. The OTG state machine is required to hook DCD's `UDEVICE` event block and use UEVENT_RESET events to reset this flag. This is in the standard core DataPump because SET_FEATURE is a chapter 9 command (although the feature was added in the OTG supplement)

Table 8. USB DCD Events significant to OTG

Event	Description
UEVENT_SUSPEND	*The OTG layer must translate this into a_bus_suspend or b_bus_suspend, depending on the current state. This can be done while processing USBPUMP_IOCTL_USBPHY_REGISTER_DCD (for discrete phys) or during DCD registration (for phys*

Event	Description
	that are integrated into the target silicon).

As Table 8 shows, the OTGCD implementation must hook itself into the `UDEVICE`'s event-node queue, so that it can detect SUSPEND events and generate the appropriate transitions to the FSM.

17.2 OTG Finite State Machine Implementation

In order to support the widest variety of OTG transceivers, and to make it easier to validate our implementation against the spec, the OTG FSM is implemented separately from the control logic that integrates the FSM with the DataPump and with the transceiver.

The OTGCD implementation is normally provided by the transceiver support code. As part of the transceiver's initialization code, it should allocate a `USBPUMP_OTGFSM` instance, and call `UsbPumpOtgFsm_Initialize_V2()` to initialize it.

Part of the FSM is a set of state variables, which closely track the variables in [USBOTG20].

The transceiver code is responsible for updating the following variables in the OTGFSM:

Table 9. OTGFSM bus-state inputs from Transceiver

Name	Ref	Summary	Comments
`a_conn`	6.6.1.5	B-device sees that A-device is connected	Set by the transceiver, but the transceiver must consider the OTG FSM state in order to do this properly.
`a_sess_vld`	6.6.1.6	Vbus is valid for A-session (as a peripheral)	Set by the transceiver by sensing voltage on Vbus. This can be updated independent of current state, just tracking comparator output.
`a_srp_det`	6.6.1.7	SRP detected by A device	Set by transceiver whenever it detects an SRP. The FSM will clear this automatically. Realistically, this should only be set while in a_idle state.
`a_vbus_vld`	6.6.1.8	Vbus is valid for A host operation	Set by transceiver by sensing voltage on Vbus. This can be updated independent of current state, just tracking comparator output.
`b_bus_resume`	6.6.1.10	Remote wakeup detected while in **a_suspend** state	Set by transceiver code when the transceiver, the root hub, or the host controller detect a resume signaled by the B-device. (Root hub or host controller use IOCTLs to signal this to the transceiver code.) The FSM will clear this automatically.

Name	Ref	Summary	Comments
`b_bus_suspend`	6.6.1.11	The A device has detected that the bus is suspended while controlled by the B device (in **a_peripheral** state)	Set by the transceiver code when a bus suspend is detected. If the suspend is detected by the HCD, and this is an external transceiver, the HCD must send use an IOCTL to set this variable.
`b_conn`	6.6.1.12	When acting as an A device, a B-device has connected to this port.	There are many conditions on how the transceiver logic must handle this; see the referenced section in the specification.
`b_se0_srp`	6.6.1.13	Port in b_idle state is ready to issue SRP	This is cleared by the FSM on entry into **b_idle** state. It must be set by the transceiver logic when the port has been in b_idle with SE0 present for more than TB_SE0_SRP milliseconds (2 ms). If the transceiver doesn't support SRP, then this need not be implemented.
`b_sess_end`	6.6.1.14	B session has ended	Set and cleared by transceiver code based on the sensed voltage of Vbus. This can be updated independent of current state, just tracking comparator output.
`b_sess_vld`	6.6.1.15	B-session Vbus is valid	Set and cleared by transceiver code based on the sensed voltage of Vbus. This can be updated independent of current state, just tracking comparator output.
`id_float`	6.6.1.16	ID pin is floating	Set and cleared by transceiver code based on the sensed state of the ID pin. Shall be FALSE when id_mid is TRUE, or when a Mini-A plug is inserted. Shall be TRUE otherwise.
`id_mid`	N/A	ID pin is "mid-impedance"	Set and cleared by transceiver code based on the sensed state of the ID pin. Shall be TRUE when a car-kit plug is inserted; shall be FALSE otherwise. Note that this definition is different than the raw output from the ISP1301 transceiver. For the ISP1301, id_mid should be calculated based on `/ID_GND * /ID_FLOAT`. This definition allows for simpler code in the common case where car-kit is not supported.[1]
`a_bus_resume`	6.6.1.3	K state detected on the bus while in **b_wait_acon**	Set by transceiver code based on detecting a K state while in the **b_wait_acon** state; cleared by

[1] The OTG FSM doesn't try to manage the interface when in carkit mode; that management logic must be provided by external code.

Name	Ref	Summary	Comments
		state.	FSM on entry to **b_wait_acon** state.
a_bus_suspend	6.6.1.4	Operating as a B-peripheral, we detected that the bus is suspended	Set by transceiver when a suspend (more than 3 ms of inactivity) is detected. If the DCD detects suspends, then the OTG layer should hook into the UEVENTNODE queue for the DCD, and process UEVENT_SUSPEND notifications.

Table 10. Software inputs from IOCTLs

Name	Ref	Summary	Comments
a_suspend_req	6.6.1.17	Request to suspend A host	The application uses this to signal that it wants to suspend the bus without ending the session, and clears it to resume. The OTG implementation is responsible for handling this by implementing USBPUMP_IOCTL_OTGCD_A_SUSPEND_REQ.
a_set_b_hnp_en	6.6.2.1	Tell FSM that HNP has been enabled for our peer.	USBDI arranges for this to be set after successfully enabling HNP. The OTG implementation handles this by implementing USBPUMP_IOCTL_OTGCD_ENABLE_HNP.
a_bus_drop	6.6.1.1	Tells FSM that the application wants to drop Vbus	The upper level application arranges for this to be set or cleared using USBPUMP_IOCTL_OTGCD_A_BUS_DROP.
a_bus_req	6.6.1.2	Requests a session as a host (as an A device)	The upper-level application arranges for this to be set or cleared using USBPUMP_IOCTL_OTGCD_B_BUS_REQUEST.
a_clr_err	6.6.4	Clears a Vbus error	The upper level application sets this as a one-shot using USBPUMP_IOCTL_OTGCD_A_CLR_ERR.
b_bus_req	6.6.9	Requests a session as a host (as a B device)	The upper-level application arranges for this to be set or cleared using USBPUMP_IOCTL_OTGCD_B_BUS_REQUEST. This implies b_sess_req. The host will have to enable HNP, and then disconnect, for this to push the device all the way to state **b_host**.
b_hnp_enable	N/A	Indicates that the B-device has received the HNP enable request	The upper level application arranges for this to be set using USBPUMP_IOCTL_DCD_HNP_ENABLED.

Name	Ref	Summary	Comments
`b_sess_req`	N/A	Requests a session as a device (as a B device)	The upper level application arranges for this to be set using `USBPUMP_IOCTL_DCD_SESSION_REQUEST`. This will cause an SRP if the device is currently in state **b_idle**.

A default IOCTL handler is built into the OTGFSM that will handle the IOCTLs appropriately.

18 Sample Host/Device/Transceiver Interactions

This section describes how things work with an Arc core connected to a Philips ISP1301 transceiver. At the low level, the transceiver is only connected to the Arc Core via USBRCV, D+, D- and nOE. ISP1301 SPEED and SUSPEND are grounded, and Vbus is driven from the ISP1301 (if at all). All OTG switching and control needs to be done in the 1301.

18.1 States of the transceiver

If we are running in simplified OTG state, then the transceiver is in one of the following states. This state machine does NOT support OTG HNP and SRP.

DISABLED	The GLOBAL_POWER_DN bit is set in the ISP1301. The DCD should have been sent a RESET followed by a cable-disconnect message Only enter IDLE when we receive a IOCTL directing the transceiver to become active..
IDLE	The ISP1301 is enabled. In this state, if ID is low, we have a host cable connected. If ID is open (ID_FLOAT is set), then we might have a device cable attached. Enable the ID_FLOAT, ID_GND, VBUS_VLD interrupts. Clear SPEED_REG. Set SUSPEND_REG. If ID_FLOAT is set and VBUS_VLD is clear, then stay in idle. If ID_FLOAT is clear, turn on Vbus, go to HCD_CHK_DEVICE. If ID_FLOAT is set and VBUS_VLD is set, go to DCD_ENA
HCD_CHK_DEVICE	Enable ID_GND, ID_FLOAT, DM_HI and DP_HI interrupts. Enable DM_PULLDOWN and DP_PULLDOWN. When an interrupt occurs: If ID_GND is not set, return to IDLE. If DP_HI is set (and not DM_HI) then signal root hub that we have a full-speed device attach (USBPUMP_USBPHY_HCD_EVENT_GOT_DEVICE) – clear SUSPEND_REG and go to HCD_GOT_DEVICE IF DM_HI is set (and not DP_HI) then signal root hub that we have a low-speed device attach (USBPUMP_USBPHY_HCD_EVENT_GOT_DEVICE), and clear SUSPEND_REG and set SPEED_REG. Goto HCD_GOT_DEVICE if DM_HI && DP_HI then we have an error – do a debug print, and return to

	HCD_CHECK_DEVICE.
HCD_GOT_DEVICE	Disable DP_HI, DM_HI interrupts. Stay in this state until HCD sends us an IOCTL. If IOCTL is "USBPUMP_IOCTL_USBPHY_SUSPEND_PORT", then set SUSPEND_REG, goto HCD_SUSPEND If IOCTL is "USBPUMP_IOCTL_USBPHY_RELEASE_PORT" then goto IDLE If we get ID_FLOAT interrupt, then cable has been unplugged; go to HCD_UNPLUGGED, and notify root hub of change (USBPUMP_USBPHY_HCD_EVENT_CABLE_DETACH).
HCD_UNPLUGGED	This is a temporary state – we come here while the HCD is responding to an HCD_UNPLUGGED message. Stay here until USBPUMP_IOCTL_USBPHY_RELEASE_PORT is received. USBPUMP_IOCTL_USBPHY_SUSPEND_PORT should be failed.
DCD_ENA	Clear DM_PULLDOWN and DP_PULLDOWN. Based on last request from DCD for "enable pull-up", set DP_PULLUP or clear DP_PULLUP. Clear SUSPEND_REG. Set SPEED_REG (we're always full speed device). Enable SESS_VLD == 0 interrupt and ID_FLOAT ID_GND interrupts. If see ID_GND=1 or ID_FLOAT=0, send "USBPUMP_USBPHY_DCD_EVENT_END_SESSION" to DCD, go to DCD_UNPLUGGED If see "USBPUMP_IOCTL_USBPHY_RELEASE_PORT", go to IDLE
DCD_UNPLUGGED	This is a temporary state – we come here while the DCD is responding to a DCD END_SESSION message. If we see "USBPUMP_IOCTL_USBPHY_RELEASE_PORT", go to IDLE

18.2 API for HCD to transceiver

18.3 ISP1301 Register Settings

18.3.1 Mode Register 1

The bits in this register are initialized as shown below.

UART_EN	reset to 0 (by USBPHY)
OE_INT_EN	reset to 0 (by USBPHY based on platform config)
BDIS_ACON_EN	manipulated dynamically for OTG
TRANSP_EN	clear to 0 (initialized by USBPHY based on platform config. Possibly manipulated based on platform information and transceiver mode control)
DAT_SE0	clear to 0 (by USBPHY based on platform config)

SUSPEND_REG	manipulated dynamically by HCD and DCD
SPEED_REG	set to 1 while in DCD mode. Manipulated dynamically by HCD based on speed of device in root port in HCD mode

18.3.2 Mode Register 2

This bits in this register are used as shown below.

EN2V7	statically initialized based on Vbat power level.
PSW_OE	clear to 0 (set ADR/PSW to input). Controlled by USBPHY based on platform config
AUDIO_EN	clear to zero (by USBPHY based on platform config)
TRANSP_BDIR1	clear to zero (by USBPHY based on platform config)
TRANSP_BDIR0	clear to zero (by USBPHY based on platform config)
BI_DI	set to 1 (by USBPHY at init time based on platform config)
SPD_SUSP_CTRL	must be set to 1 (by USBPHY at init time based on platform config)
GLOBAL_PWR_DN	initialized to 0 by USBPHY. Manipulated by USBPHY based on power management IOCTLs

18.4 OTG Control Regiser

The bits in this register are used as shown below.

VBUS_CHRG	Used during SRP (Vbus based)
VBUS_DISCHRG	Used during SRP (Vbus based)
VBUS_DRV	Set to 0 during device mode. Set to 1 in HCD mode to turn on Vbus. Set to 0 when waiting for a connection.
ID_PULLDOWN	Clear to zero (by USBPHY during init based on platform config), meaning that ID is controlled externally
DM_PULLDOWN, DP_PULLDOWN, DM_PULLUP, DP_PULLUP	Manipulated by USBPHY based on device/host state.

18.5 OTG Status Register

The bits in this register are used by the OTGCD to handle session management

B_SESS_VLD	indicates that Vbus is high enough to start a session. It's used by the HCD (via root hub) to detect over-current and faults.
B_SESS_END	indicates that Vbus is no longer present, and ends a DCD session. The corresponding interrupt should result in an event going to the DCD.

18.6 Interrupt Source Register

The bits in this register are used by USBPHY and OTGCD to detect events.

CR_INT	DP pin is above car kit interrupt threshold. Initially, this will be ignored (no car kit support)
BDIS_ACON	indicates that the B device has become the host after HNP
ID_FLOAT	ID pin is floating – used for detecting unplug or plug of mini-A or accessory connector
ID_GND	ID pin is grounded – used for detecting unplug or plug of mini-A or accessory connector
DM_HI, DP_HI	D-minus pin is high. Used for tracking data-pulsing SRP
SESS_VLD	"session valid" signal – indicates that a cable is attached and Vbus is active. (in device mode)
VBUS_VLD	"Vbus valid" – indicates that Vbus is > 4.4V. This should be used when functioning as a device to detect that a cable is attached to a live host.

19 Scheduling

MCCI's USBDI does most of the scheduling work, and communicates its decisions to the HCD via the USBPUMP_HCD_PIPE structure. The HCD must then create the appropriate schedule entries that match what USBDI has done; otherwise the HCD runs the risk of over-subscribing the bus. For periodic transfers that happen every frame, it's pretty easy for the host controller to put things into the schedule (apart from USBDI having to make assumptions about what the host controller can really do – there's no guarantee that a host controller can really issue transfers every frame).

But there's a missing piece of information that USBD and the HCD have to agree on. If periodic transactions for a given pipe happen less than once every frame, then USBDI and the HCD have to agree on how often a transaction will be scheduled, and in which frames.

For example, if a transfer happens once every other frame:

1. USBD need to know whether the HCD can support "every other frame" transfers.

2. If so, USBD needs to be able to specify (to the HCD) whether the transfer is to occur on the even or odd frame. [Otherwise, all of the even-numbered frames would fill up with

periodic transfers, possibly causing failures when plenty of bus bandwidth was still available.]

More generally, USBDI needs:

1. To know the topology of the scheduling tree within the HCD

2. To tell the HCD where to put a given pipe within the scheduling tree.

For reasons of testability, USBDI offers the HCD a limited number of options. The simplest is two choices: either all periodic transfers are scheduled every frame (which works OK for a small number of endpoints and devices); or else a binary tree is maintained. The root of the tree is for "every frame" periodic transfers; the next layer is for "every other frame" periodic transfers and has two nodes: one for transfers in even-numbered frames, and one for transfers in odd-numbered frames. This process repeats. Generally, levels (starting from 0) have 2^n entries, where entry 0 represents transactions that occur for (micro)frames where frame-number mod 2^n == 0, entry i represents transactions for (micro)frames where frame-number mod 2^n == i, and so forth.

On full-speed host controllers, this tree is normally 6 layers deep, from 2^0 through 2^5. We could allow more or fewer layers rather trivially, but it costs RAM for bandwidth tracking. A tree with 6 layers has $2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0$ possible transaction patterns, and therefore needs 2^6-1 cells to track total bandwidth. If USBD uses USHORTs, that's 126 bytes.

With hardware like OpenHCI or EHCI controllers, nothing prevents software from establishing endpoint service trees that are not binary. For example, periodic transfers might be scheduled either every frame or every 8 frames. This would save memory for queue heads, but would effectively make the test matrix of USBDI extremely large. Furthermore, a binary decomposition is common, but not necessary at the hardware level; it is possible (if scheduling is done in software) to use a decimal decomposition where there are (for example) 1000 queue heads rather than 1024. But supporting that degree of flexibility seems wasteful in developer and test time; and being able to deal with trees that branch in arbitrary radix will necessarily slow down the common case of binary branching.

So USBDI simply assumes that the HCD will use a binary tree to manage bandwidth. USBDI allows the tree to have adjustable depth. Public cells in the HCD structure inform USBDI of the depth of the scheduling tree.

To communicate info from USBD to HCD, the HCD needs (at least) to know the frame number offset at the layer within the tree. The HCD can compute the layer within the tree by taking the $\log_2$ of the transfer interval. However, for convenience, USBD passes the level in the `USBPUMP_HCD_PIPE`, in the field `PeriodicBucket`. The level is passed in the field `PeriodicLevel`.

USBDI observes the following constraints.

This holds the offset in the buckets at the assigned level for this endpoint. This must always be in the half-open interval

$[0, 2^{\texttt{PeriodicLevel}})$

or equivalently in the closed interval

$[0, 2^{\texttt{PeriodicLevel}} - 1]$

It is convenient to lay out the schedule buckets as a linear table[2]:

[0] level 0

[1] level 1, offset 0; [2] level 1, offset 1

[3] level 2, offset 0; [4] level 2, offset 1; [5] level 2, offset 2; [6] level 2, offset3

[7] level 3, offset 0; [8] level 3, offset 1, ..., [14] level 3, offset 7

[15] level 4, offset 0; [16] level 4, offset 1, … [30] level 4, offset 15

… and so forth

Then the linear index K of the schedule bucket for entry (level, offset) is given by:

$K(level, offset) = 2^{level} - 1 + offset$

or equivalently:

K(level, offset) = (1 << level) + offset – 1

Given a linear index value K, it's convenient to be able to find the parent directly. A simple formula may be derived as follows:

Kparent(level, offset) = (1 << (level-1)) + (offset >> 1) – 1

Note that

K + 1 = ((1 << level) + offset)

so

(K + 1) >> 1 = ((1 << level) + offset) >> 1

= (1 << (level-1) + offset >> 1

so if K > 0,

Kparent(K) = ((K + 1) >> 1) – 1

[2] This is one of the few tables that is more easily understood and discussed using 1-origin rather than 0-origin indexing; but habit dies hard.

20 Historical / Reference Information

20.1 Initialization Discussions

The new DataPump initialization scheme is modeled on TrueTask's, in that the initializer simply iterates over a vector of pointers to functions, in three phases. Initialization proceeds by calling all the functions sequentially. There are three phases. First phase is intended to allow modules to set defaults; second phase is intended to allow modules to install themselves; third phase is run "after the system is up".

Object initialization proceeds as a depth-first traversal of a tree. At each level, a function is called which creates a corresponding object, taking as its input the current pointer-to-parent object, and a pointer to the node.

It would be nice to be able to represent this level of interconnection as a data structure. Since some applications of the DataPump chip driver are for a given chip, we would like the data structure to be able to omit things that are not needed (both code and data). This implies that the link to the code will have to be via the init vector plus libraries. Of course, in a system of this complexity, we'd also like to be able to test all the valid configurations – this implies a large number of demo apps.

If we want an abstract representation, the HCD is going to need a number of pieces of info. This is similar to the "wiring". To support systems in which the system dynamically assigns port numbers, the resource assignments should be held separately from the "wiring info" which is defined at design time.

So each HCD instance will need a resource description object, containing:

1. Bus handles
2. Port addresses
3. Interrupt assignments
4. Anything else that might be changed dynamically at run time by something to which the HCD is subordinate.

Each HCD instance will also need a wiring table, defined as at present, except that the interrupt assignments presently in wiring tables will move to the resource description object. The wiring table will contain the probe pointer for the HCD instance. The probe function's job is to confirm that the HCD is both present and to be used. (Hence all wiring tables need to be derived from a common object).

For sanity, all of these will be passed around in a top-level object. This object will contain a pointer to the rest of the HCDs.

With the change in possible topologies, the DataPump initialization scheme has to change slightly. The description of a "port" in the port init vector can no longer describe something

that results in a UDEVICE; instead, evaluating the entry must yield TRUE if some hardware was found, FALSE otherwise. (This way, the clients can still know at startup time if any hardware was probed, and report an error if none was found.)

Our first stage is to probe the hardware. We use a probe list, consisting of a sequence of probe instructions:

- a pointer to the structure that defines this object's code cluster – must be derived from a simple type USBPUMP_MODULE_CLASS table, beginning with a probe function and a setup function.
- a bus-handle (optional)
- an I/O port (optional)
- an arbitrary pointer to the configuration data for this instance (format defined by the object) to be used by the probe and setup functions.

For UDEVICEs, we define a USBPUMP_MODULE_CLASS_UDEVICE object, which contains a probe function, a setup function, and a pointer to the UDEVSWITCH [or we can make the UDEVSWITCH derived from the USBPUMP_MODULE_CLASS].

For host controllers, we define a USBPUMP_MODULE_CLASS_USBPORT

The new port init vector has the following scheme:

XXX	Need to fill in the host controller initialization scheme. I'm trying to come up with a simple scheme: run a vector, and get back a forest of objects. But the current UDEVICE initialization scheme is blocking my progress. No major problem, just need another few hours of thought.