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 User's Guide

Engineering Report 950000066
Rev. P
Date: 2016-08-03

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.

Document Release History

Rev. A	1998-07-31	Official release.
Rev. B	1999-04-26	Partial update. Released for customer review.
Rev. C	1999-12-08	Partial update. Checked structures, functions.
Rev. D	2000-10-12	Updated structures, functions and added new interrupt system interface.
Rev. E	2002-03-15	Added overviews, build process.
Rev. F	n-a	Not released.

Rev. G	2003-02-15	Update for V1.8 release.
Rev. H	2009-01-26	Updates and revisions.
Rev. I	2010-10-14	Changed document numbers to nine digit versions. DataPump 3.0 Updates.
Rev. J	2010-10-15	Corrected Table and Figure numbering problems. Corrected reference bookmarks problem.
Rev. K	2011-09-26	Added source code disclaimer.
Rev. L	2012-09-13	Updated Chapter 7, Chapter 13, Chapter 12 for V3.10 release.
Rev. M	2014-03-04	Added new members in struct TTUSB_PLATFORM. Added new debug mask. Used UDATAPLANE_OUTSWITCH instead of UDEVICE_OUTSWITCH. Added UENDPOINT_CAN_AUTO_REMOTE_WAKEUP and UENDPOINT_CHECK_AUTO_REMOTE_WAKEUP macro. Added new members in TTUSB_UDEVICE and TTUSB_UINTERFACESET. Added CreateAbstractPool Function. Added new Device Related Function.
Rev. N	2015-10-14	Various clarifications. Add documentation of descriptor-based isochronous APIs. Clarify UDEVICE, UENDPOINTSWITCH, and use modern names. Fix various typos and update links.
Rev. P	2016-08-03	Various clarifications.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. Introduction

This manual introduces the MCCI USB DataPump, a portable USB firmware development kit for adding USB device support to embedded products based on 16-, 32- and 64-bit processors.

1.1 What's the MCCI USB DataPump?

The MCCI USB DataPump is a complete, portable firmware package for implementing high-performance USB host and peripheral devices. It encapsulates low level USB hardware-control code and high-level device class support in a comprehensive framework, allowing firmware development engineers to focus on the target application, rather than the details of USB interfacing. It is portable across operating systems, CPUs, endianness, hardware platforms, USB silicon, compilers and development platforms.

1.1 References

Reference	Title	Version	Comments
[USB2]	USB Serial Bus Specification	2.0	The normative specification for low speed, full speed, and high speed USB hosts, hubs, and devices
[USB3]	USB Serial Bus Specification	3.1	The normative specification for SuperSpeed (5 Gbps Gen1 and 10 Gbps Gen2) hosts, hubs, and devices.

1.2 Glossary of Terms

Table 1. The Terms Used in This Spec

Term	Meaning
ACM	Abstract Control Model Protocol, a subclass of the Communication Deivce Class (CDC) of USB-IF.
bsdm4 utility	The DataPump software utilizes a macro processor tool called *bsdm4* for processing source code files written in a language called *m4*. The DataPump software uses the *m4* language to write assembly code that is independent of processor type. The output of the *bsdm4* tool is the assembly language for a specific Target CPU. This approach is only used for hardware level interface code that is difficult or impossible to do at the C language level in a platform independent manner.

Term	Meaning
bsdmake utility	A version of the standard Unix *make* utility. *bsdmake* is a program designed to simplify the maintenance of other programs. The DataPump software uses *bsdmake*, running in the TTK Unix-like shell, to build all versions of object files, link files, resource files, and executables.
build	The process of creating part or all of an executable object from source code. The standard DataPump distribution uses the *bsdmake* utility to perform all builds. However, when integrated with an RTOS distribution, the DataPump is often built using the target operating system's build system.
Catena	The MCCI Catena cards are development tools that provide a USB device support for testing embedded USB implementations in a PC-based development environment.
CDC	USB Communication Device Class. This is a formal USB specification that specifies how to implement general communication protocols.
Chapter 9	Refers to Chapter 9 of [USB2] and [USB3], which define all of the required operations of a USB device..
Class driver	An instance of "driver class" for a particular "device class".
Composite Driver	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. In the DataPump environment, 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.
DataPump	Refers generally to the software development kit contained within.
DCD	Device Controller Driver. The software component that provides low-level access to the specific Device Controller in use.
device driver	Hardware interface level software or firmware that controls the operation of a hardware device.
DFU	Device Firmware Upgrade. A protocol of USB for the programmatic upgrading of firmware on USB devices.
Final Hardware	A version of Target Hardware representing the production version or representative of the production version. Software developed on Final Hardware should be identical to that delivered to a customer of the final product.
gcc	GNU C compiler

Term	Meaning
gdb	GNU debugger.
HCD	Host Controller Driver. The software component that provides low–level access to the specific Host Controller in use.
HDVT	MCCI's Host Driver Verification Tool. It is designed to assist developers in testing host drivers for the MCCI USB DataPump Embedded Host and On-The-Go products.
HID	Human Interface Device.
HIL	Hardware Interface Layer. This refers to software that interfaces directly with hardware devices (i.e. the lowest level of software). The DataPump software abstracts this level of software to be independent of USB interface hardware.
HNP	Host Negotiation Protocol of USB OTG.
loopback [protocol]	The DataPump device stack comes with a supplied loopback protocol that connectes pairs of OUT and IN pipes. Whatever the host sends to the OUT pipe of a loopback pair, the DataPump echoes back on the corresponding IN pipe. The Loopback protocol is used to verify correct installation, and to validate that hardware is operating properly.
m4	The DataPump software utilizes a macro processor tool called *bsdm4* for processing source code files written in a language called *m4*. Because of the lack of standardization of assembly language syntax for ARM and other processors, MCCI uses the *m4* language to write assembly code that is independent of the toolchain being used. The output of the *bsdm4* tool is the assembly language for a specific toolchain. MCCI also uses m4 to write register definition files that can be used to generate both C header files and assembly-language definition files.
make	See *bsdmake. make* is a program designed to simplify the maintenance of other programs.
makefile or .mk file	A type of text file that follows conventions used by a *make* utility. See *bsdmake.*
MIB	Management Information Base.
MSC	Mass **Storage** device Class.
MTP	Abbreviation for Media Transfer Protocol. It supports the transfer of music files on digital audio players and movie files on portable media players.
OTG	Abbreviation for USB On-The-Go.
protocol	Refers to both the software and the specification to support a particular methodology of communication across a communication medium. For the DataPump, protocol

Term	Meaning
	usually refers to the software and specification of the use of the USB bus for a particular device type. The Mass Storage Bulk-Only Transport (BOT), for USB Mass Storage devices, is an example of a protocol across USB.
shell file (.sh file)	An input "batch" file to a Unix command line prompt application (similar to a .bat file in MS-DOS). When loaded by a Unix shell, the ".sh" extension is often omitted.
SNMP	Simple Network Management Protocol.
Target	Normally refers to the combination of Target Hardware and Target Software/Firmware.
Target CPU	The CPU used on the target hardware (e.g. Samsung ARM-7, Motorola 68302, etc.). This is different from the host computer's CPU which is usually a PC or workstation.
Target Hardware	The hardware that will run the DataPump software.
Target Operating System	The operating system that runs the DataPump software once loaded onto the target hardware.
Target Software/Firmware	The software name referred to when discussing software running on the Target Hardware. This is different from operating system device drivers that may be required on the Host Computer to communicate with the Target device.
Target USB device controller/DCD	The USB device controller/DCDr that runs on the Target Hardware. This is different from the host computer's USB host controller/HCD.
Thompson Toolkit (TTK)	A third party software package used by the DataPump build system. The TTK provides a Unix-like shell environment and executables used during the build, compile, link, and debug phases of development. If your target is not using the MCCI build system, or if you are using Cygwin, this may not apply to you.
Transaction Translator(TT)	A functional component of a high-speed USB hub. The Transaction Translator responds to special high-speed transactions and translates them to full/low-speed transactions with full/low-speed devices attached down downstream facing ports.
USB	Universal Serial Bus.
USBD	USB Driver, the generic term for the USB host stack module.
USBIOEX	MCCI USBIOEX application was designed to assist developers in testing firmware for USB devices.
USBRC	USB Resource Compiler. This application was developed by MCCI to aid in the creation of USB resource descriptor files required by all USB devices.
WMC	Wireless Mobile Communications, a subclass of Communication Device Class (CDC) of USB-IF.

1.3 Target Audience

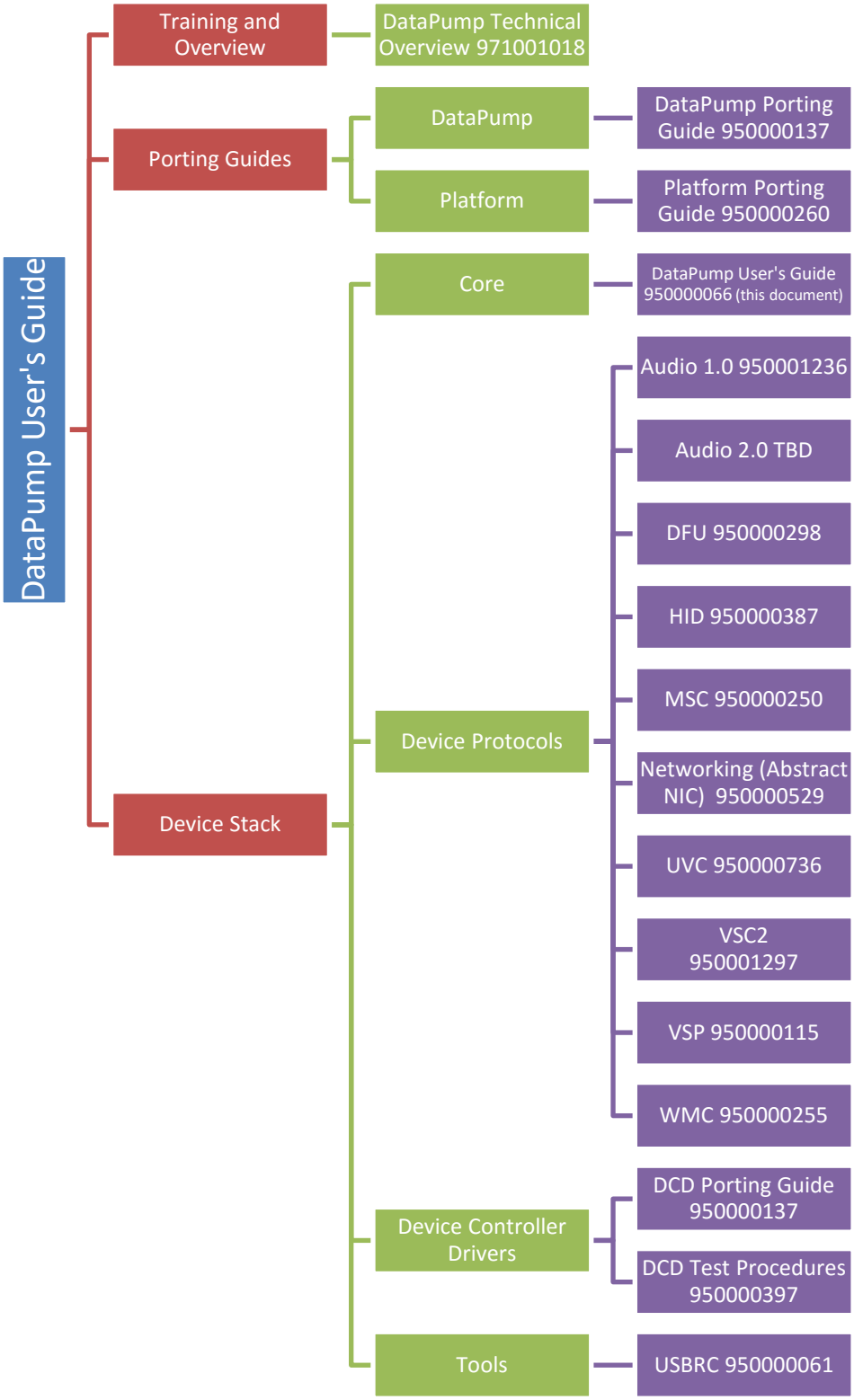
The *MCCI USB DataPump User's Guide* is for embedded system engineers who wish to use the MCCI USB DataPump to create a specific USB device.

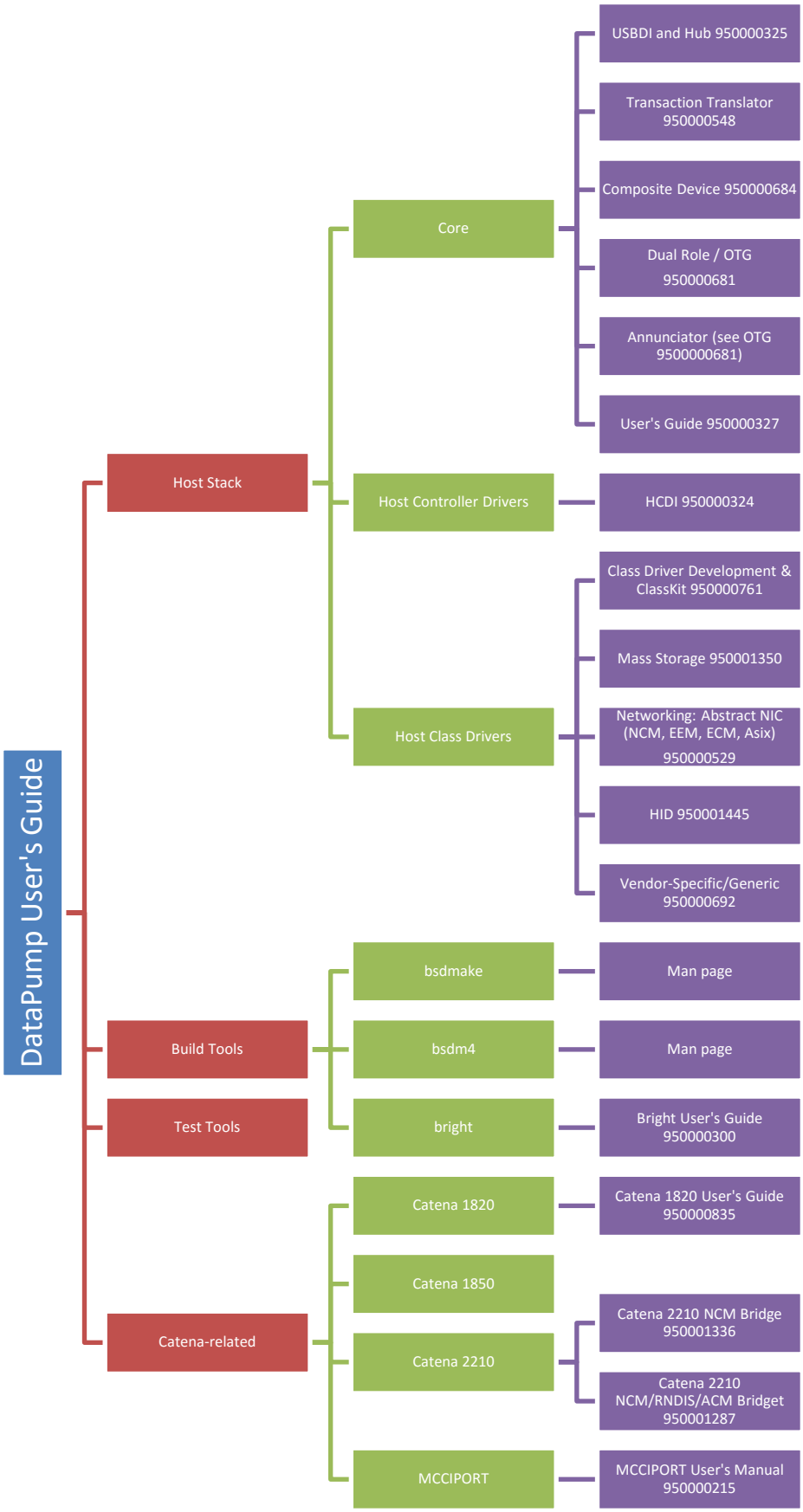
We assume you are familiar with the general concepts of developing embedded systems, including cross-compilation, and development in C.

1.4 Documentation Tree

Because of the high degree of modularity of the DataPump, the documentation is necessarily divided into several independent manuals. This manual, the "*MCCI USB DataPump User's Guide,*" is the primary manual in the set of manuals for the MCCI USB DataPump. Figure 1 graphically shows the various manuals related to the MCCI USB DataPump.

Figure 1. DataPump Documentation Tree




DataPump User's Guide
Host Stack
Core
USBDI and Hub 950000325
Transaction Translator 950000548
Composite Device 950000684
Dual Role / OTG 950000681
Annunciator (see OTG 9500000681)
User's Guide 950000327
Host Controller Drivers
HCDI 950000324
Host Class Drivers
Class Driver Development & ClassKit 950000761
Mass Storage 950001350
Networking: Abstract NIC (NCM, EEM, ECM, Asix) 950000529
HID 950001445
Vendor-Specific/Generic 950000692
Build Tools
bsdmake
Man page
bsdm4
Man page
Test Tools
bright
Bright User's Guide 950000300
Catena-related
Catena 1820
Catena 1820 User's Guide 950000835
Catena 1850
Catena 2210
Catena 2210 NCM Bridge 950001336
Catena 2210 NCM/RNDIS/ACM Bridget 950001287
MCCIPORT
MCCIPORT User's Manual 950000215

1.5 Related Documentation

The following is a list of documentation relevant to the DataPump

1.5.1 MCCI USB DataPump Documentation

Table 2. MCCI USB DataPump Documentation

Title	Document Number	Description
MCCI USB DataPump User's Guide	950000066	This document
MCCI USB Resource Compiler Users Guide	950000061	Describes the USB DataPump Resource Compiler.
MCCI USB DataPump Porting Reference Manual	950000137	How to create a new DCD
MCCI USB DataPump Platform Porting Guide	950000260	How to port the DataPump to a new platform
MCCI USB DataPump Mass Storage Protocol User's Guide	950000250	Describes the Mass Storage protocol implementation
MCCI USB DataPump WMC Protocol User's Guide	950000255	Describes the implementation of the WMC family of protocols (CDC modem, OBEX, Device Management)
MCCI USB DataPump Virtual Ethernet Protocol User's Guide	950000290	Describes the Virtual Ethernet protocol implementation
MCCI USB DataPump New DFU Protocol User's Guide	950000298	Describes the DFU protocol implementation

1.5.2 MCCI USB Support Software for Windows

Table 3. Documentation for MCCI USB Support Software for Windows

Title	Document Number	Description
Generic Drivers -- OEM Reference Manual	950000108	This document is the reference manual for the MCCI Generic Drivers for Windows. These drivers are used with USBIOEX for testing basic USB function.
USB I/O Exerciser -- Users Guide	950000075	This document is the reference for USBIOEX, a Windows application that is used in conjunction with the MCCI Generic Driver for testing basic USB function.
W-CDMA Bus Drivers -- Functional Specification	950000185	This document is the reference manual for the MCCI enhanced USB composite driver for Windows. The manual is included for completeness. The driver is primarily relevant to customers who are preparing USB modems, but can be useful when handilng complex multi-configuratoin devices.

2. DataPump Product Overview

2.1 Overview

The MCCI USB DataPump is a portable software package for implementing USB devices.

It is an object-oriented, portable firmware package written in ANSI C. It contains code that implements the core USB protocols, interfaces for abstract platform and device controller driver objects, an automatic code generator, and a basic library of function-specific protocols.

The MCCI DataPump product line includes the following components:

- Core libraries for USB device implementation
- Device controller drivers specific to a given register model
- Operating system abstraction layers for specific target operating systems

- Device class protocols such as Audio, Video, Still Image, Mass Storage Class, HID, CDC ACM, NCM

- Reference applications that operate with the core DataPump software and specific protocols to implement complete USB devices, for purposes of example

- MCCI Catena products

The MCCI Catena products allow testing embedded USB device and host implementations in a PC-based development environment. The Catena card provides the PC with a USB device controller, allowing the PC to emulate a USB device (instead of being the host) to facilitate device development. They can provide everything developers need to prototype USB device firmware, USB host firmware and dual role USB firmware solutions in Windows systems. For example, the Catena 1820 development platform enables development and testing of USB firmware for the Renesas R8A66597 device controller..

Customers can add support for new processors, operating systems, and compilation environments following procedures given in the *MCCI USB DataPump Porting Reference Manual* and the *MCCI USB DataPump Platform Porting Guide*.

The DataPump is primarily intended for creating a high-performance multi-function USB platform that can be used to support a family of target applications, scaling from full-speed USB devices to SuperSpeed USB devices and high-speed USB On-The-Go. The DataPump product includes all of the base building blocks ready to run including support for multiple USB device controllers, multiple operating systems, multiple build/compilation environments, and multiple protocols.

The DataPump also serves as a platform for several other MCCI products, including the MCCI TrueTask USB Embedded USB Host Stack.

2.2 DataPump Base Product vs. Application & Protocol Add-ons

2.2.1 Components Overview

The DataPump includes build tools, the GNU C compiler and debugger, a USB resource compiler for the creation of USB descriptors, and source code for all of the Target CPU, Target USB interface, Target Operating System, and Cross Compilation Development environments as described in the previous section. In addition, it comes with complete code for the support of Chapter 9 of the USB 1.1, USB 2.0 and USB 3.1 Specifications, including enumeration, configuration, and data transportation.

2.2.1.1 Code

- Header Files

- Device Stack

- MCCI DataPump Core (Implements Chapter 9 of the USB2.0/3.0 Spec)
- Device Class Protocol Modules (Implement the relevant device class specs)
- API Modules
- Device Controller Driver (DCD)

  This is low-level hardware interface code

- Host Stack (optional)
  - Host Controller Driver (HCD)
  - Host Controller Driver Framework (HCDKIT)
  - USBD
  - Hub Driver
  - Composite Driver
  - Transaction Translator (TT) Support.
  - Class Driver (Basic HID, MSC, Generic)

2.2.1.2 Tools and Build System

- Building Tools
- Thompson Toolkit
- BSDmake
- Bright
- USBRC
- Debug and Verification Tools
- Test Applications

To completely implement a USB device, some additional components are also required. The main three components that are required beyond the base product are:

A. protocol layer

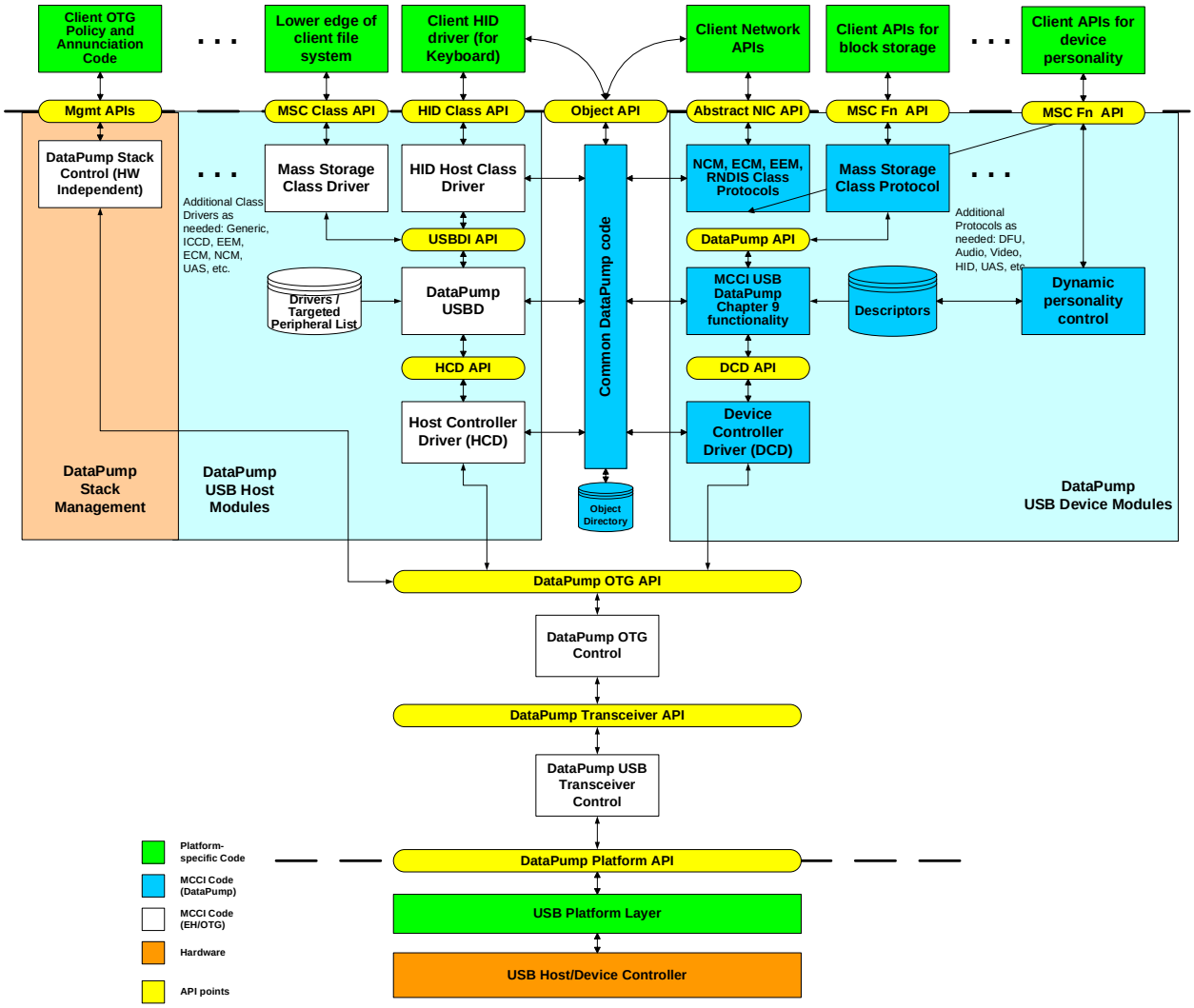
B. application layer

C. destination port software

MCCI offers several standard protocol and application layers for common use cases. Users can also create of custom protocol and application layers. Generally speaking, some code must be modified or written by the customer, in particular the destination port software.

Figure 2 shows the overall software/hardware architecture of the DataPump device stack.

Figure 2. Architecture of MCCI USB DataPump device/OTG stack



2.2.2 Roadmap to Product/Documentation Usage for a Specific Use Cases

The DataPump is a large and flexible product that can be applied to at least four general use cases. The best way to read the documentation depends very much on the intended use case.

Use Case #1: Creating a USB device based on an MCCI-supplied porting layer, DCD, class protocols, and applications:

1. Prepare the information requested in *Section 3.2 Information to Gather Before Beginning Development* and read background material in Section 3.
2. Follow the steps in Section 3.4.4 DataPump Cross Compile Build/Debug Process to create a build environment for the target application.

3. Modify the copied application code in the target application directory as needed by referencing the corresponding *MCCI USB DataPump Protocol Reference Manual* for the protocol of interest. For non-protocol, hardware and operating systems functions, refer to Sections 7 through 13 in this manual.

Use Case #2: Creating a USB device based on an MCCI-supplied porting layer, DCD, and class protocol modules, while creating a new application:

1. Prepare the information requested in section 3.2 Information to Gather Before Beginning Development, and read background material in Section 3.
2. Follow the steps in section 3.4.4 DataPump Cross Compile Build/Debug Process to create a build environment for the specific new application, including a shell application.
3. Modify the shell application code in the new application directory to meet the specific needs, referencing the corresponding *MCCI USB DataPump Protocol Reference Manual* for the particular protocol. For the general facilities of the DataPump, refer to Sections 7 through 13 in this manual.

Use Case #3: Creating a USB device implementing a new device class protocol not supported by MCCI, but using MCCI-supplied porting layer and DCD:

1. Prepare the information requested in Section 3.2 Information to Gather Before Beginning Development, and read the background material in Section 3.

2. Follow the steps in section 3.4.4 DataPump Cross Compile Build/Debug Process to create a build environment for the specific application.

3. Modify the protocol and application code in the new application directory to meet the specific need. For the general facilities of the DataPump, refer to Sections 7 through 14 of this manual.

Use Case #4: Porting the DataPump software to an unsupported a) USB device controller, b) CPU, c) target Operating System, and/or d) cross compilation development environment.

1. Follow the process for the previous choice that corresponds to the desired final goal.

2. Obtain the *MCCI USB DataPump Platform Porting Guide* and follow the instructions.

2.2.3 Device Class Protocol Modules

This section describes each of the MCCI created Protocol Modules that operate with the DataPump. These modules add support for specific USB Device Classes. Each Protocol Module receives the class-specific commands from the host, decodes the commands, and translates them into a set of class specific function calls appropriate to the abstract semantics of the class. For example, CDC WMC Modem and CDC WMC OBEX interfaces each have a different command

set over USB; but they are both translated into a common WMC API that allow upper-edge clients to treat them in a common way, if the system engineer finds this appropriate. Each Protocol Module has a corresponding *MCCI USB DataPump Protocol Reference Manual* that specifies it in detail.

Protocol Modules generally operate by creating DataPump objects (see section 8 MCCI DataPump Object System) at initialization time. These objects serve as the instance objects for the protocols, and export API methods that are used by upper-edge code.

For example, the CDC Ethernet Protocol attaches to the appropriate `UINTERFACE` and `UPIPE` structures exported by the DataPump. It receives the USB commands specific to the Ethernet Control Model of the USB Communications Device Class (CDC) specification, and either implements them directly or translates them into Ethernet-specific operations that are issued to the upper-edge client. The client is coded in terms of Ethernet frames and Ethernet-specific control plane events.

A Protocol Module is incorporated into the overall application by means of the following steps:

1. Modify the URC file to incorporate the interfaces, endpoints and descriptors appropriate to the device class.

2. If the MCCI build system is being used, modify the application's `UsbMakefile.inc` build control file to reference the public header files and libraries for the protocol module.

3. Modify an application's Protocol Initialization vector to include one or more `USBPUMP_PROTOCOL_INIT_NODE` elements. These elements cause interfaces of a specified kind to be bound to a specific protocol, and also cause the protocol module code to be linked into the run-time image.

4. Write "upper-edge" code that locates protocol instances of the desired kind, and uses the provided interface to operate on those instances.

The following sub-sections list the MCCI supported protocols that are either included with the base product or may be separately purchased direct from MCCI. Contact MCCI for more information on protocols that may have been created since the printing of this manual or for a quote on the creation of a custom protocol. To create a custom protocol independently, refer to the roadmap in the section 2.2.2 and study *Section* 5 *Implementing A Custom Protocol or Application.*

2.2.3.1 Loopback (loopback)

The loopback Protocol Module implements a generic debugging & hardware verification protocol. The Loopback protocol attaches to an interface with at least one IN pipe and one OUT pipe; it simply echoes back whatever is sent to it across the USB from the host. This can be used to learn the build process, verify correct hardware/software installation, and/or test the performance characteristics of a USB system.

2.2.3.2 Virtual Serial Port (*vsp*)

The Virtual Serial Port Protocol Module implements MCCI's proprietary Virtual Serial Port device class definition, which allows high-fidelity implementation of remote serial ports. This protocol module receives and decodes control-plane messages, and calls application-supplied code to perform the appropriate UART-like operations.

2.2.3.3 Device Firmware Upgrade (*DFU*)

The USB Device Firmware Upgrade class specification specifies how to upgrade firmware on a USB device in a standard, device independent manner. The DataPump DFU protocol receives these standard commands and transmits them to a higher-level application for processing.

Refer to the *MCCI DataPump Device Firmware Upgrade Protocol Reference Manual* for more information.

2.2.3.4 Human Interface Device Class (*hid*)

The USB Human Interface Device (HID) class specification specifies how to receive/send information from a wide variety of devices, ranging from standard input devices like keyboards and mice to device specific applications such as Uninterruptible Power Supplies and LCD/CRT monitor controls. The MCCI HID Protocol Module serves as a general-purpose transport-layer protocol to implement the capabilities of this specification.

Refer to the *MCCI DataPump Human Interface Device Class Protocol Reference Manual* for more information.

2.2.3.5 Networking Related Protocols (*Abstract NIC*)

The DataPump's Abstract NIC mechanism allows you to use a variety of Ethernet-over-USB protocols with a single integration. The same integration is used with the host and device stack. The following class protocols are supported in both host and device mode: CDC NCM, CDC ECM, CDC EEM, and Microsoft RNDIS. When operating as a host, Abstract NIC also supports connecting to ASIX USB-to-Ethernet adapters.

MCCI has created three optional networking related protocols to provide great flexibility in the creation of various device and bridge applications.

Refer to the corresponding protocol reference manuals for more information.

2.2.3.6 USB Mass Storage Class (*usbmass*)

The Mass Storage Class protocol is a part of the USB specification for implementing devices that send/receive bulk data, such as an external USB hard drive. The *usbmass* protocol receives commands and data and transmits them to a higher-level application for processing. See *Section 2.2.4.4 Mass Storage Class Demo (mscdemo)* for a description of an application using this protocol.

Refer to the *MCCI DataPump Mass Storage Class Protocol Reference Manual* for more information.

2.2.3.7 USB CDC WMC Subclasses (*wmc*)

The MCCI WMC Protocol Library, in conjunction with the MCCI USB DataPump, provides a straightforward, portable environment for implementing CDC-ACM and CDC WMC ACM compliant AT-compatible modem devices and modem-like devices over USB using the following protocols:

- USB CDC 1.1 Abstract Control Model (ACM) protocol [USBCDC] and the USB WMC 1.0 ACM
- USB WMC 1.0 Object Exchange (OBEX) and
- USB WMC 1.0 Device Management protocols [USBWMC].

The MCCI WMC Protocol Library can be used to create a standalone device, or can be combined with other protocols to create multi-function devices.

2.2.4 Demo Applications

This section describes MCCI's demo applications for the DataPump. Contact MCCI for more information on applications that may have been created since the printing of this manual or for a quote on the creation of a custom application. To create a custom application independently, refer to the roadmap in the previous section and *Section* 5 *Implementing A Custom Protocol or Application*.

2.2.4.1 Loopback

The loopback application uses the loopback protocol module to implement a simple test application.

This application can be used with MCCI's generic drivers and MCCI's USBIOEX to perform data integrity testing and basic integration testing.

2.2.4.2 Virtual Serial Port Demo (*vspdemo*)

The Virtual Serial Port Demo (*vspdemo*) application turns a USB device into a USB to Serial converter. It uses the Virtual Serial Port protocol to receive messages across USB and echoes them on a Target hardware serial port.

2.2.4.3 Device Firmware Upgrade Demo (*dfudemo*)

The Device Firmware Upgrade Demo (*dfudemo*) is a sample application using the *dfu* protocol to receive standard USB specification protocol commands to update the firmware of the Target Hardware. This software may be modified for use with any hardware.

Refer to the *MCCI DataPump Device Firmware Upgrade Application Reference Manual* for more information.

2.2.4.4 Mass Storage Class Demo (*mscdemo*)

The Mass Storage Class protocol is a part of the USB specification for implementing devices that send/receive bulk data such as an external USB hard drive. The mscdemo application uses the usbmass protocol to create a simulated file system in the form of a RAM disk. This sample application can be modified to send/receive the data transfers to any type of storage device.

Please contact MCCI to purchase this optional DataPump application. If already purchased, refer to the *MCCI DataPump Mass Storage Class Application Reference Manual* for more information.

2.2.4.5 WMC Demos

Several WMC demos are provided. All the completed demos use os/none and simply return data received from the host back to the host. wmcdemo loops data received on a given USB function back to the same function; wmc2pdemo cross-connects data received from one WMC / ACM modem function to a second on the same device.

3. Developing with the USB DataPump

3.1 Overview

This section provides DataPump background and use information which intends to:

1. prepare the user for efficient use of the DataPump,
2. provide background information on cross compile environments in general and the DataPump environment specifically, and
3. walk the user through the details of the DataPump development process.

3.2 Information to Gather Before Beginning Development

The base DataPump product comes with a development environment and support for a general USB device. To complete the device, the developer will also need to select the protocol and application that will require support.

3.2.1 Planning Final Hardware

If the time has come to move/develop software onto Final Target Hardware, some information must be collected to configure the DataPump Software for the device:

1. Target CPU
2. Target USB device controller
3. Target Operating System
4. Cross compiler
5. Download and Debug methods
6. Hardware interfaces other than USB to be supported.
7. Memory and Interrupt map of the Target Hardware

For 1, 2, 3, and 4, the specific names/versions of the hardware/software must be gathered and cross checked with the list of supported configuration combinations in *Appendix A DataPump Supported Configurations.* If this combination of targets is not supported directly it will be necessary to either (a) purchase, if not done so already, the *MCCI USB DataPump Porting Reference Manual* or (b) contract MCCI directly for support or outsourcing of the port of the particular target software.

For 5, multiple issues require a decision. To note, this information is partially tied to the decision for #4 since the download/debugging software environment is usually tied to the cross compiler. In is imperative these two areas are tightly integrated. The default methodology assumes the use of two serial ports on the Target hardware to act as a console and debug port respectively.

For 6, It is necessary to compile the names/versions of the hardware interfaces (or custom interfaces) and to include any console/debug I/O devices required to develop and debug the firmware.

For 7, the DataPump Software needs to know where each of the hardware resources reside to operate correctly. The user must also configure their hardware specific resources to work in tandem with the core DataPump hardware resources. Once the information from #1 through #6 has been compiled, it is necessary to layout a memory and interrupt map to aid in the configuration of the DataPump Software.

3.3 Cross Compilation Environment Overview

There are significant differences between developing "normal" software that runs natively on the computer where it was developed and developing dedicated software/firmware for a new hardware platform, especially a low cost hardware device such as a USB device.

The cost of manufacturing a hardware device is an important issue that drives the profitability of the hardware product. Standard computer resources required for software development such as a "PC-Class" CPU, hard disk, ample RAM, a video port, a keyboard, and a mouse, are often less readily available to firmware developers. These hardware devices usually contain lower-end processors that are not compatible with a standard PC, with limited resources that do not allow

for support of a development environment. These limitations require the use of a Host/Target cross compilation software development environment. The MCCI DataPump Firmware Development Kit is such a development environment.

In a Host/Target development environment, the Host is the computer that is used to develop and build the executable code. The Target is the hardware that the developed code will be executed on.

3.3.1 Cross Compilation vs. Native Windows Development

Table 4 gives a step-by-step comparison of the key similarities and differences of developing software in a PC-native software development environment and cross compilation environment.

Table 4. PC-Native vs. Cross Compile Software Development Process

"Normal" or PC-Native Software Development Environment	Cross Compilation Software Development Environment
Development Host Computer and Target Execution Hardware are the same computer or at least of the same type (e.g. X86 running Windows)	Development Host Computer and Target Execution environment are different platforms. Target hardware usually a dedicated piece of hardware with limited computer resources (e.g. RAM, I/O, etc.)
Development Tools reside and are run on Host computer	same
Executables created on Host computer	same
Executables use Host computer operating system calls and are in the binary language of the Host processor	Executables use Target Hardware platform operating system calls (if any) and are in the binary language of the processor on the Target Hardware .
Debug/development versions of the executable (i.e. during creation/testing) run in place on Host computer	Debug/development executables must be sent to the Target Hardware, usually involving a download port and protocol agreed upon between the Host and Target hardware.
Native debug tools help walk through executable running on Host computer	Depends on Target operating system chosen. Some have "remote" debuggers that run on the Target hardware and use a serial port back to the Host computer to control this debugger. In other cases, the developer must write all of their own debug code to send messages to the debug serial port
Final executable (shipped with product) is a file(s) that is installed from a distribution disk (or downloaded from the Internet) onto a computer of the same class as the Host computer. In the case of hardware drivers, this executable is	Final executable usually "burned" as firmware into a non-volatile hardware resource on the final hardware (e.g. EPROM, Flash). In some cases, this executable may be installed on a Host computer and "soft loaded" (i.e. at boot

"Normal" or PC-Native Software Development Environment	Cross Compilation Software Development Environment
usually loaded as part of the operating system booting process	or initialization time). This firmware is the only software to run on the hardware
New versions of a Final executable (i.e. updated over time after a customer has purchased the product) are common and usually sent in the same fashion as the original distribution	New versions of a Final executable are rare in the case of "fixed" non-volatile storage devices (i.e. those that use burn-once EPROMs) since updating this firmware requires the replacement of a hardware chip. In the case of Flash or re-programmable EPROMs, a download protocol must be developed to update the chip(s). In the case of a "soft loaded" architecture, replacing the file on the host updates the executable automatically

3.3.2 Taking into Account the Firmware Developer when Designing the Final Hardware

The Hardware Developer must take the Firmware Developer into account when designing the Final Hardware. To verify/test and debug firmware bugs on the Final Hardware, the Firmware Developer requires a way to quickly iterate firmware revisions and "see" the inner workings of their software on the Final Hardware. Consideration should be given to this need up front.

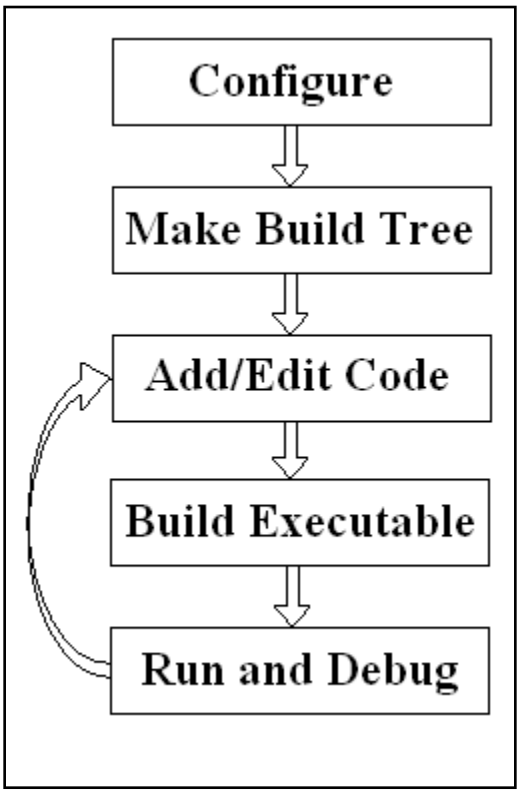
The developer of the Final Hardware should strive to keep the hardware resources to a minimum in the final product. This can be in conflict with the firmware developer's need for hardware resources not necessary in a fully debugged Final Hardware product. For example, developing firmware requires the ability to get code to the Target hardware and get debug messages out and may require additional RAM on the Target hardware (for loading of a debugger, debug version of the target operating system, etc.). These hardware resources would not be required once the code is fully tested and operational.

3.4 DataPump Development Process Background

3.4.1 DataPump Development Process Overview

The MCCI DataPump development process is similar to most development processes with a few added steps. The primary steps are depitcted in Figure 3 and includedevelop: setting up the configuration, making the build tree, adding/editing code, building the executable, and running/debugging the executable.

Figure 3. DataPump Software Development Process



The last three steps are nearly identical to any software development process. The difference occurs in the first two steps. The DataPump was created to handle a large wide range of Target Operating Systems, Target Compilers, and a variety of hardware configurations, including different CPUs and USB device controller register sets. As such, the DataPump software needs to be configured or setup to build the particular combination of software that is correct for a particular Target hardware and particular application.

From this configuration set, a build tree is created that includes the directory structure, files, and configuration parameters of a particular Target hardware and application. The build tree is created once per application type. After this process is complete, all of the source and build files are ready to use. Code can be edited and executables built, run, and debugged.

3.4.2 DataPump Usage of Third Party Tools

To support a large combination of Target CPU, Target compiler, and Target Operating Systems, as well as both Windows and Unix host systems, MCCI created the DataPump software using a common development toolset. This toolset is based on a Unix-like build structure and uses several third party software tools, as depicted in Table 7

Table 5. Build Tools used by the DataPump

Third Party Software	Purpose
Thompson Toolkit	Provides Unix-like shell (line command prompt) and many Unix-like commands.
Cygwin command line utility	Alternative to the Thompson toolkit, supported by MCCI for development, but not shipped with MCCI's standard environment.
bsdmake	MCCI Standard build tool, derived from NetBSD's pmake. *bsdmake* is a program designed to simplify the maintenance of other programs. It takes a text file input and processes the commands contained within. See the *MCCI/index.html* for a link to further information. This utility runs from the command line and is used to perform all compilations, linking, and executable builds.
bsdm4 M4 interpreter	*Bsdm4* is a macro processor that is used to setup a common macro language. For Arm and other processors, MCCI delivers assembly language code in a portable form, which is then translated at compile time into the assembly language used by the target toolchain.
bright (portable script interpreter)	The *bright* scripting language is used to simplify cross platform programming within bsdmake. It is embedded in bsdmake, and is available as a separate programming language. It is equivalent in capability to awk or lua.

3.4.3 Unix-like development Process vs. Native Windows Development Process

In both Unix and non-Unix environments, the DataPump software uses a Unix-like code organization and building mechanism along with the use of some additional tools. If the user familiar with Microsoft Visual C, but not familiar with development under Unix (Solaris, Linux, etc.), Section 3 provides important information to aid in understanding. Table 8 compares corresponding capabilities of the Windows Visual C development process to the DataPump development process.

Table 6. Visual C vs. DataPump Development Process

Windows Visual C Development Process using 3rd Party Libraries	MCCI DataPump Development Process
PC-native code development (not for embedded environments)	Cross compilation code development process (see *Section 3.3.1* for further information). This is where the majority of the differences exist.
Creation of a "project" to describe all files in the application. Usually fixed for a single purpose taking one project per combination of code.	Pre-built *makefile* structure that is configured by the end-user but dynamically pulls in appropriate files into the build process (i.e. much better for handling very flexible configurations).
Integrated compile, run, and debug functions	Separate applications for compile, run, and debug due to cross compilation and target hardware nature of project.
Standard C files	Standard C files with some assembly type language (i.e. m4) for some hardware interface code.
3rd Party defined function calls and data structures	MCCI (3rd party) defined function calls and data structures

3.4.4 DataPump Cross Compile Build/Debug Process

Two methods exist for the creation of a build environment for an end application:

1. Using a supplied configuration utility called DataPump Config, or
2. By manually copying/editing appropriate files to create a shell environment to work from.

The DataPump Config utility quickly sets up the correct build environment that most accurately depicts the desired development effort. This utility should be used in most cases. In some situations that include more advanced configurations or modifications of the DataPump software, a manual process may be required.

3.4.5 Using DataPump Config Utility

The DataPump Config utility requests the information gathered as outlined in Section 3.2 Information to Gather Before Beginning Development and uses it to tailor a build environment to the developer's specific needs. The result is a ready to use, full source code base and build environment.

4. USB Overview and DataPump Implementation

This section provides an overview of the workings of USB in general, and the MCCI USB DataPump in particular. The process consists of the following steps:

- Select the protocols to be used, and determine from the protocol manuals which features need to be placed in the URC file.
- Create the .urc file that describes the device and its descriptors.
- Select the MCCI device class protocols to be used with your device.
- Create the additional required supporting .c files and UsbMakefile.inc files
- Use the makebuildtree script to create a build directory for your target
- Compile and link these files with the core MCCI USB DataPump, and with any additional required protocol modules.
- Write the glue code to connect the data streams on the USB to the data streams in the device. This code includes the logic that calls the initialization entry point at the appropriate time.
- Using the MCCI USB DataPump and the supplied loopback application, demonstrate the functionality of a prototype board.

4.1 Introduction to USB Device Architecture

All USB devices follow a standard architecture, outlined in Chapter 9 of the USB Core Specification.

- A ***device*** is composed of one or more ***configurations***; only one configuration can be selected at a time. That configuration is called the ***active configuration***. Each configuration within a device is identified by a unique numerical index, which is its ***configuration number***. Configuration number 0 is a special ***default configuration***. When the default configuration is selected, the device is not operational; bus-powered devices in the default configuration must obey special power restrictions.

- Each configuration is in turn composed of one or more ***interfaces***. All the interfaces in a given configuration are available if the configuration is selected. Interfaces are normally used to represent a single function of a multi-function device. However, in communications devices with multiple logical data circuits, one interface is normally used for each logical data circuit. Each interface is identified by a unique numerical index, which is its ***interface number***.

- Each interface, in turn, has one or more ***alternate settings***. Just as only one configuration can be selected in a device at a given time, only one alternate setting can be selected in a given interface at a given time. The selected alternate-setting is called the ***active interface setting***, or just the ***active interface***. Each alternate

setting for a given interface is identified by a unique numerical index, also called the ***alternate interface setting.*** For each interface, alternate interface zero is the default setting for that interface.

- Each alternate setting assigns certain properties to ***endpoints***, which are the fundamental addressable units on the USB.

- Because endpoints are hardware objects, and endpoint *settings* will change based on the current configuration and alternate settings, USB literature commonly calls the combination of an endpoint and its settings a ***pipe***. A pipe is ***active*** whenever its alternate setting and configuration are selected by the host. Similarly, an endpoint is active whenever one of its associated pipes is active. Multiple pipes might use the same physical endpoint; but within a given configuration and alternate settings, an endpoint can only be associated with one active pipe at any given time.

- For device control purposes, every device has a dedicated ***default pipe***, which is always associated with ***endpoint zero*** of the device.

The exact structure of the device is represented to the host via the USB device descriptor and configuration descriptors. The host uses this information to load the appropriate device drivers, and to determine how to route control messages to the appropriate object within the device.

Device drivers are normally associated with "functions" on a device. A device with only one function is called a "single function" device; a device with more than one function is called a "multi-function" device. Normally all the USB resources within a device for a given function are controlled by a single device class protocol instance in the DataPump.

Control messages are always sent via the default pipe on endpoint zero. However, these messages are then routed to one of four different layers in the device:

1. **The device as a whole.** Messages at this layer are used for configuration and to ask the device about its properties.

2. **An active interface or interface set.** Messages at this layer are used to select the active interface setting, and to control the operation of the active interface. These messages are addressed using the interface number.

3. **An active endpoint.** Messages at this layer are used to clear error conditions on the endpoint, or to perform protocol-specific operations.

4. **Some "other" (unspecified) location.** None of the above.

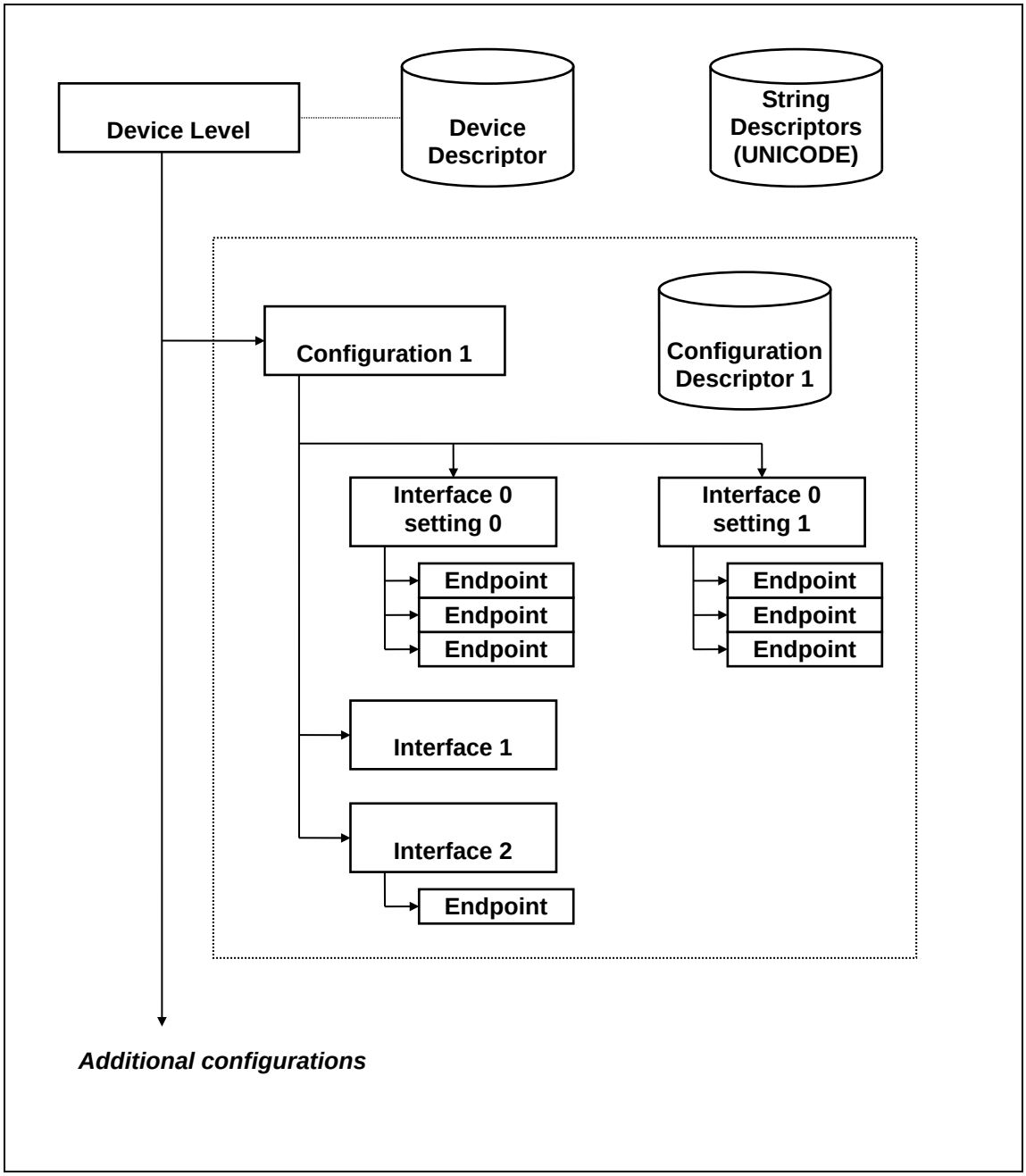
Of these destinations, only the first three are commonly used.

In addition, provisions are made for devices to carry natural-language descriptive text, which the host system can present to the user even if the host system doesn't recognize the device. This text can appear in many different languages, as chosen by the designer. Unfortunately, the text must

be prepared in , in the byte order used by Intel systems, which can make it a little awkward to prepare. However, the DataPump's USBRC tool makes it easy to prepare these strings, even if the target compiler doesn't support Unicode in the USB byte order, and in UTF-16LE encoding.

Figure 4 is a schematic diagram showing many of these features.

Figure 4. USB Device Architecture



4.2 Introduction to USB Data Transport Methods

The USB specification defines four kinds of endpoints, each of which has its own link-level protocol. A given physical endpoint might change types, depending on which configuration and alternate interface is selected. However, once the host selects a configuration and alternate interface settings, the type of the endpoint is fixed until the host changes the configuration.

The four types are:

1. **Control.** Control endpoints use a transaction-oriented protocol. The host sends a ***setup*** packet, identifying the operation to be performed. Then the host either transmits additional data packets to the device, or requests response data packets from the device, as selected by a flag bit in the packet. Control-endpoint data transfers are interlocked and positively acknowledged. Control endpoints are inherently bi-directional.

   Endpoint zero of every USB device is permanently configured as a control endpoint. Endpoint zero is the default pipe which is always available and is available in all configurations. Nothing can happen on a USB peripheral until it is enumerated and configured. These operations take place on the default pipe.

2. **Bulk.** Bulk endpoints are used to transport data that is not time critical. Data delivery is ***reliable***; packets are delivered in order, and the rate of the sender is automatically matched to the rate of the receiver. However, USB does not guarantee how quickly bulk data will be moved, and it is moved only when there is no time-critical data to be moved.

3. **Isochronous.** Isochronous endpoints are used to transport data that is time critical, and which is useless if it is delivered late. Data delivery is ***best effort***; packets are delivered in order, but packets that cannot be delivered on-time are discarded. The receiver is expected to be able to somehow substitute "reasonable" data if packets are dropped. The receiver must be able to keep up with the offered data rate, or data will be discarded. When the host computer opens an isochronous device, the required USB bandwidth for any isochronous endpoints will be allocated to that device.

   Isochronous endpoints were intended to be used to transport audio or video data, for which missing data is not as bad as late data. Despite this, some devices use isochronous endpoints to transport normal data. In this case, the device and the host driver have to agree on a protocol on top of the basic isochronous protocols, to provide rate matching and error recovery.

4. **Interrupt.** Interrupt endpoints are used to transport time-sensitive data with more error checking than is available for Isochronous endpoints.

   Interrupt endpoints are intended to be used to transport asynchronous information. Like Isochronous endpoints, bandwidth is assigned to Interrupt endpoints, guaranteeing a certain minimum polling rate. However Interrupt endpoints use a protocol that includes retries and data error recovery, so the effective data transfer rate may be less than the pollng rate (if there are retries), and there is no upper bound on latency.

4.3 The MCCI USB DataPump Device Model

The DataPump models a given device as a tree.

- At the root of the tree is a structure representing the USB device, the "***UDEVICE***".

- Under the root is a collection of structures, representing each possible configuration; each structure is called a "***UCONFIG***". The UDEVICE contains a pointer to the collection of UCONFIGs, and also to the active UCONFIG.

  The UDEVICE also contains pointers to the tables of descriptors associated with the device.

- Under each UCONFIG is a collection of structures, one for each interface. The DataPump calls these structures ***UINTERFACESET****s*, because each one is a collection ("set") of alternate settings.

- Under each UINTERFACESET is a collection of structures, one for each alternate setting for this interface. (Even alternate setting zero, the default, is treated as an alternate setting.) Each structure is called a ***UINTERFACE***. Each UINTERFACESET contains a pointer to the collection of UINTERFACEs, and also a pointer to the active UINTERFACE within that collection.

- Under each UINTERFACE is a collection of structures, one for each endpoint associated with the alternate setting. These structures are called ***UPIPE****s*.

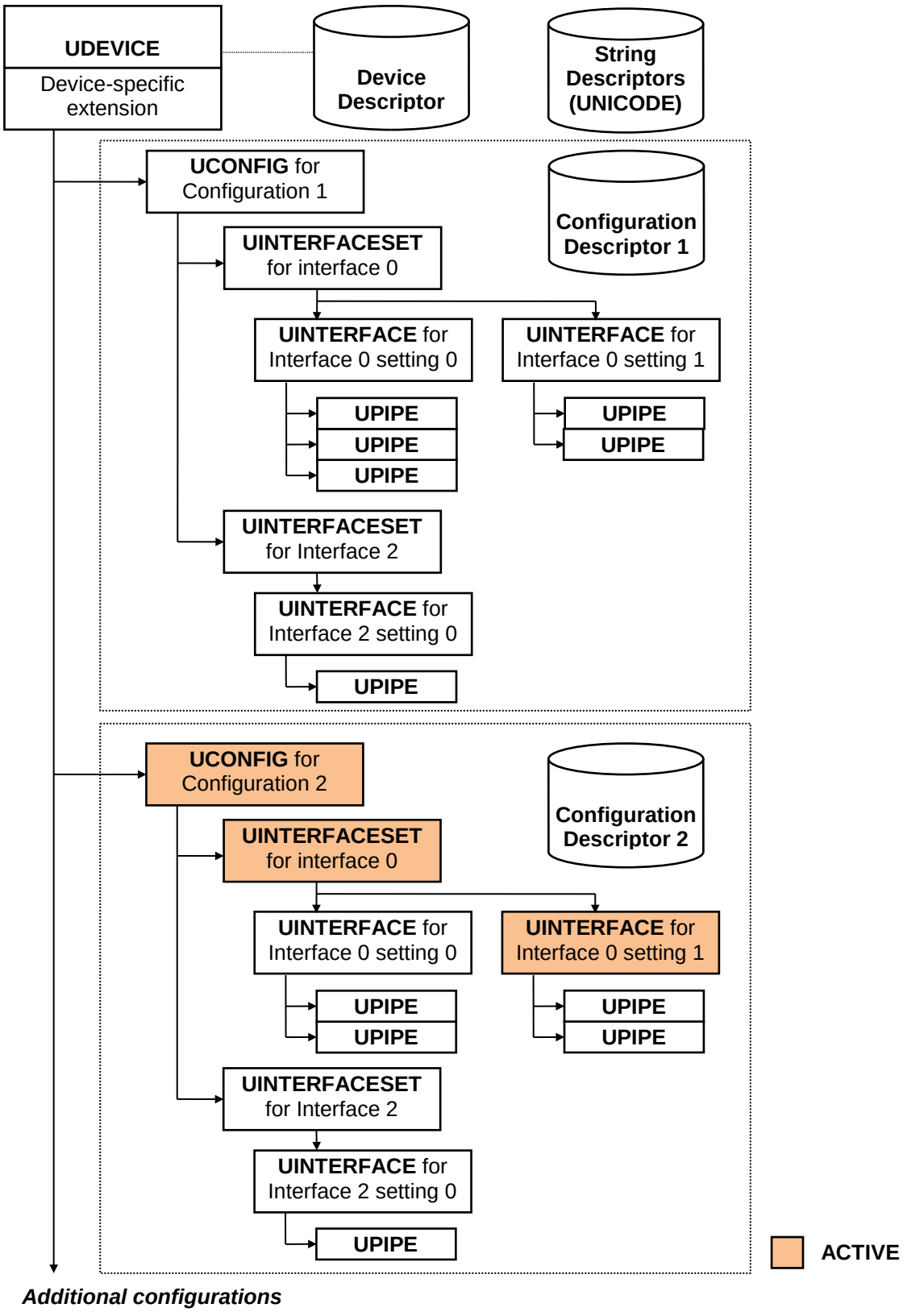
- Each UPIPE contains information about the desired mode for the hardware endpoint in the alternate setting, based on information provided in the ".URC" file. (Refer to ER950000061 for details on .URC files) In addition, each UPIPE points to a ***UENDPOINT*** structure that models the hardware resources for that endpoint.

- UENDPOINTs are abstract data structures that contain two kinds of information: information used by the portable code (for queuing and control), and information used by the hardware-specific code. Normally, UENDPOINT is treated as the base type for the actual structure that is used by the hardware-dependent layers.

  For example, the port of the DataPump for the FT900 USB controller declares an UENDPOINT_FT900DCI structure that represents a UENDPOINT, with additional, hardware-specific information appended to it. So a given block of memory, representing an endpoint, will be viewed in two ways: as a UENDPOINT by the portable code; and as a UENDPOINT_FT900DCI by the FT900-specific code. The same is true for the UDEVICE structure, the port declares a UDEVICE_FT900DCI structure that has additional port specific information appended to the UDEVICE structure.

Figure 5 is a schematic of these data structures.

Figure 5. USB DataPump Abstract Device Model



4.4 MCCI USB DataPump Device Operations

The USB DataPump API has two fundamental interfaces that are used by applications or protocol modules.

4.4.1 Data Transfer

Applications transfer data to or from the host in a very traditional way, by issuing data transfer requests. (This is sometimes called an "active" API, because the application actively calls the DataPump to cause data transfers to occur.)

The basic DataPump interface is asynchronous and non-blocking. An application prepares a transfer request, called a ***UBUFQE*** (short for "buffer queue element"), which contains the following information:

- the UPIPE to be used for the transfer,
- a description of the buffer of data to be transferred,
- some flags, which control the fine details of how the information is to be moved, and
- a pointer to a function to be called when the operation finishes.

The application then calls ***UsbPipeQueue()***. UsbPipeQueue links the buffer into the queue for the endpoint associated with the pipe, performs any initialization required, and returns to the caller.

Later, when the data has been transferred (transmitted or received), the USB DataPump calls the application's call-back function to notify the application that the transfer is finished.

This approach works well for single speed (full speed) devices. For dual-speed devices (full speed plus high speed, or full speed/high speed/SuperSpeed), a more elaborate mechanism called the UDATASTREAM is used to represent all the various UPIPEs that might be used for data transfer, depending on the actual speed of connection. This mechanism also helps simplify the design of multi-personality devices. UDATASTREAMs are implemented as a layer on top of the basic UPIPE / UsbPipeQueue() mechanism.

4.4.2 Control

USB control operations function differently. Instead of the application calling the DataPump directly, applications or protocol modules ***register*** event-processing functions with the DataPump. (This is sometimes called a "passive" API, because the application passively waits to be called whenever events occur.) The DataPump allows multiple event-processing functions to be registered. In addition, event-processing functions are associated with specific levels in the device tree; the DataPump automatically demultiplexes events and delivers them only to the appropriate level.

There are two general classes of USB events.

- Events which result from Chapter 9 processing are processed by the core DataPump. Notifications are then issued to the affected levels of the device tree. Examples of these events include suspend, resume, bus reset, configuration selection, and interface setting selection. If the events in this class have no significance to the device firmware outside the DataPump, the user need not provide event handling for them; the USB DataPump will handle them appropriately on its own.

  Many Chapter 9 events are handled entirely by the DataPump without notification to the protocols. These events include getting descriptors, clearing features, and so forth.

- Default-pipe operations that are beyond the scope of Chapter 9 are passed to the appropriate level of the device tree for processing. If the event functions supplied by the application do not handle the operation, the DataPump sends an error indication back to the host, and aborts the operation.

4.4.2.1 Event Queue Processing

Both kinds of events are handled by an idle loop that polls for and responds to event notifications, as shown in Figure 6. Events can be posted or put on the queue by hardware interrupts, application requests, and firmware polls.

The USB connection is shown on the left and works with the hardware serial interface unit and function interface unit. When valid packets have been sent or received, an interrupt is generated. Rather than acting directly on the interrupt, an event is posted to the queue for later processing. Optionally, the port can check if the DataPump is idle, and if so, call the action directly from the interrupt, and only post an event if the DataPump is busy. This could improve efficiency since the event mechanism is bypassed if the circumstances allow it.

In a similar manner, the application interface, with its hardware, will eventually require some sort of data exchange with the USB side. In the Figure 8 example, the modem hardware needs to deliver received data packets or has finished sending a packet. The appropriate message is placed on the event queue.

The idle loop is specific to the operating system and platform. It simply polls the event queue for happenings and acts accordingly. On platforms with no OS, the idle loop effectively is the OS, and MCCI provides simple code of the form:

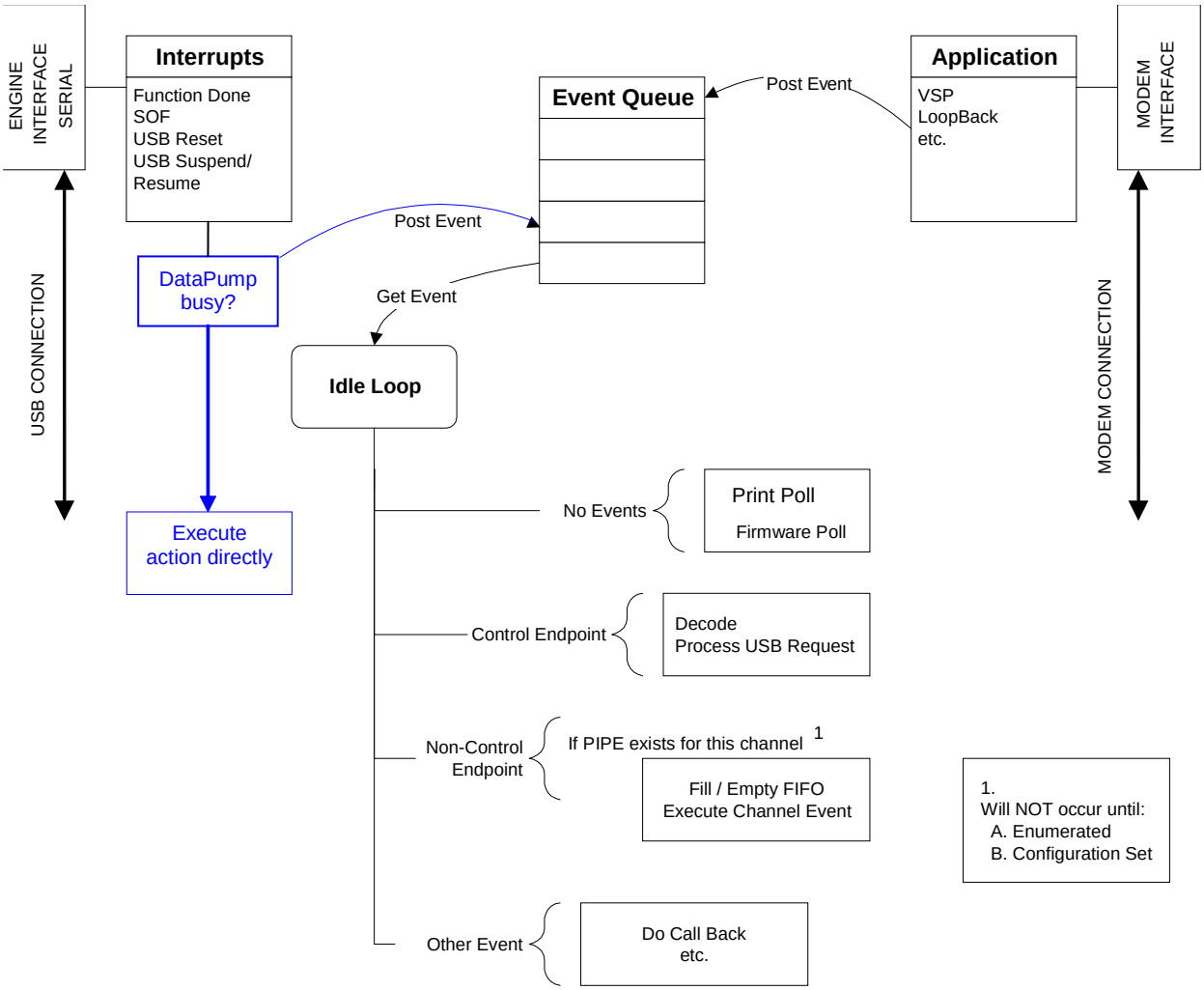
```
while (1)
       {
       UHIL_DoEvents(PUEVENTCONTEXT pevq, ULONG max_events);
       UHIL_DoPoll(PUPOLLCONTEXT ppoll, PUEVENTCONTEXT peq);
       );
```

The events that are on the queue are processed in the order that they were placed there. The most significant events are:

- Decoding and processing USB messages on endpoint 0

- Processing non-control endpoint USB happenings- buffer full/empty, etc. This will only occur after the device has been enumerated, a configuration set, and the application has set up a buffer.
- Other - call backs, reports, etc.

Figure 6. Event Queue Processing



If no events exist on the queue, then the idle or background tasks are performed. These tasks include:

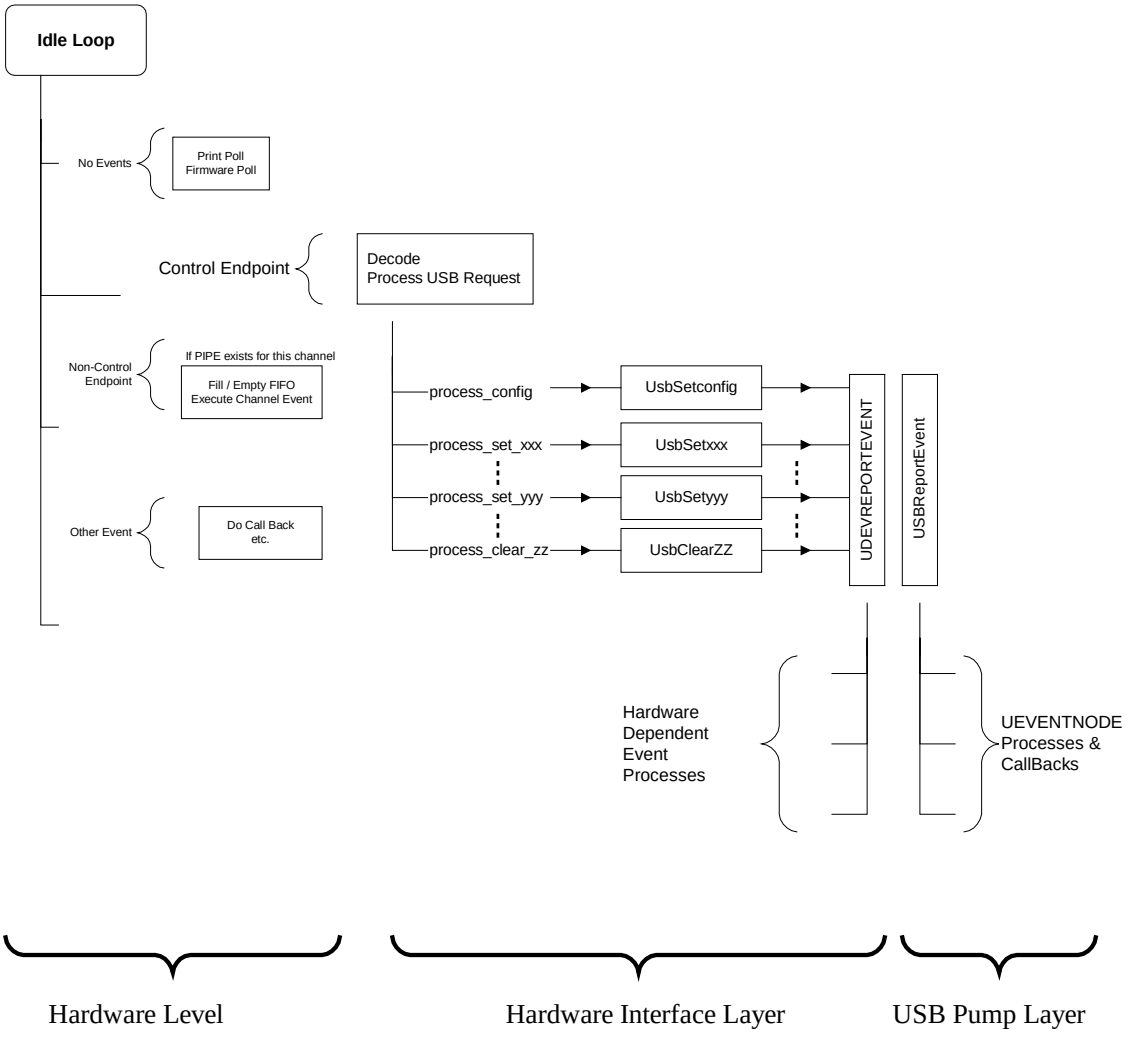
- print polling for sending diagnostic or status messages - development phase only
- firmware polling for non-interrupt hardware and application servicing
- polling for pending USB events - usually after processing normal USB notifications

The firmware polling is established when an application sets up a firmware poll function. This is the hook from the idle loop to the user application that can be used for periodic service of application hardware.

4.4.2.2 Processing the Default Pipe

The processing of control endpoint events is more extensive than most of the other events. (See Figure 7) The control endpoint software is responsible for decoding and then processing the USB packets. Any hardware specific items are addressed in the "process" functions. A call to the hardware interface layer is made so that any required actions can be done at a global level. When required, calls are made back to hardware level for further action. The call may be passed on to the DataPump layer where calls to all the pipes, endpoints, etc. may be made. These global calls to the hardware are normally done when changing or establishing configurations.

Figure 7. Control Endpoint Processing



4.4.2.3 Switches

Tables of function pointers are often used to provide the interfaces between well-defined software layers. Following UNIX kernel terminology, these tables are called "switches".

The mechanism for configuration change is in the SWITCHES. At the top level, the UDEVICE switch, defined in the structure UDEVICESWITCH, is responsible for sending messages down the chain toward the hardware level. These messages tell the appropriate level to create or destroy endpoint, pipe, interface set, interface, and configuration links. At the bottom end, each endpoint has an UENDPOINT switch, similarly defined by the UENDPOINTSWITCH structure, which is used to send information upstream to signal significant events in the USB data or other hardware happenings.

Two other switches are used. The UEVENTNODE structure **may** invoke a function when a message is sent down to change configuration or interface. The other is in the Channel Event that is used to invoke some action when a USB data packet is received on a properly configured channel.

4.5 Protocols and Device Classes

As mentioned previously, the "Chapter 9 commands" are those commands defined by the core USB specification as being required for any USB peripheral. The DataPump itself implements the Chapter 9 functionality. However, Chapter 9 compliance is not enough; it is still necessary to define how to use USB to control and activate the device.

The core specification defines a general-purpose transport protocol. This protocol can be thought of as being analogous to IP – it specifies how to move data and commands, but does not define the meaning of those data and commands. Device Class specifications define more specific commands and data formats for a variety of specific functions. The device designer uses Device Class specifications and builds upon the general protocol to implement more specific functions. The USB Implementers Forum (USB-IF) has defined a number of standard Device Class protocols that can be used as the basis for a given design. For example, modems often follow the Communication Device Class (CDC) Abstract Control Model (ACM) specification as an additional protocol layer added on top of Chapter 9.[1]

On the other hand, for some devices there is no adequate protocol defined. In thsese cases, the USB specification requires a vendor-specific protocol be defined. Although this means the device designer must provide their own drivers, which provides much more flexibility; it can even help reduce the cost of the device.

For example, RS-232 ports are not supported very well by the CDC specification. MCCI has defined a vendor-specific protocol and protocol extensions to allow RS-232 to be handled either

[1] A CDC "Abstract Control Model" modem uses two bulk endpoints to send data packets to/from the host. The host uses the control endpoint to perform control operations (dialing the phone, etc.) An interrupt endpoint is used to pass asynchronous notifications back to the host. These endpoints are collected into two "interfaces", a Communication Class interface and a Data Class interface.

as an extension to the CDC spec (allowing drivers to be shared) or as a purely proprietary protocol (to reduce the cost of the device.)

When programming a protocol, the designer must decide whether a given protocol layer is ***concrete*** or ***abstract***. Concrete layers are always at the top of the protocol stack (furthest from the DataPump code); typically they combine function calls to the next layer of the protocol stack with operations that are specific to the device. Concrete layers are clients of the lower protocol layers. For example, in a USB-to-RS-232 converter, a concrete layer would map the MCCI VSP protocol operations into equivalent operations targeting a UART.

Abstract layers are normally not at the top of the protocol stack. They typically convert the semantics of the lower protocol into different, more specific semantics particular to that layer; and they hide information about the details of the lower layer from the client. Abstract layers are a very powerful tool for creating reusable and highly structured code; however, they impose a minor overhead due to the extra function calls that are typically involved.

The USB DataPump supports both programming models. The ***loopback*** protocol supplied with the DataPump evaluation kit and with the Firmware Developer's Kit is an example of a trivial concrete protocol – data which comes from the host on one pipe is reflected back to the host on another pipe. It's also an example of a protocol that is self-configuring; the loopback protocol will correctly attach to any interface with a pair of pipes, whether the pipes are bulk or isochronous.

In more complicated devices, abstract protocols are connected to higher-layer software protocols that are part of the host operating system. For example, in a cell phone, the Wireless Mobile Communication device class instances in the DataPump are normally connected to instances of the AT interpreter in the cell phone. This connection is normally made using a generic object-oriented API using I/O Control operations (IOCTLs). This allows the connection between the DataPump and the host operating system to also be self-configuring.

4.6 MCCI USB DataPump Implementation Details

The DataPump is written entirely in ANSI C, and has a layered implementation.

- The hardware interface layer is a thin layer that provides OS- and platform-specific services. The DataPump is carefully designed to require very little from the platform.

- The device-controller driver (DCD) layer provides the code specific to the target USB device controller hardware. This allows MCCI to support numerous different register sets (such as the FT900 or the Synopsys DWC_OTG) with the same code base for the majority of the code.

- The DataPump itself implements the Chapter 9 functionality: enumeration, configuration, data transportation, and so forth. The DataPump is designed using asynchronous, non-blocking calls, and an I/O queue for each endpoint. This allows event-driven transfers, much in the style of WDM drivers.

- In addition, the DataPump maintains the Chapter 9 model of the state of the device. As interfaces and endpoints are "configured" by the host, application-specific event handlers can be called. This provides the hooks for layering multiple independent protocols on top of the DataPump.

- The DataPump provides a general mechanism for organizing and identifying protocol objects. This allows MCCI and customer-supplied code to find the facilities configured into the DataPump dynamically, and eliminates the need for conditional compiles in either.

Application modules (e.g., the loopback test, the virtual serial port module, or the CDC modem module) are layered on top of the DataPump. This connection is inherently dynamic and reentrant; so multiple instances of a CDC interface can be supported as easily as a single interface. This might be important in cable modem applications, for maintenance mode, for.

The MCCI USB DataPump is completely portable with respect to target processor byte order (little-endian vs. big-endian.)

4.7 Components of a Total USB Solution Using the MCCI USB DataPump

In the USB world, a product requires software and firmware on both the host side and the device (function) side of the wire.

Figure 8. Total USB Solution

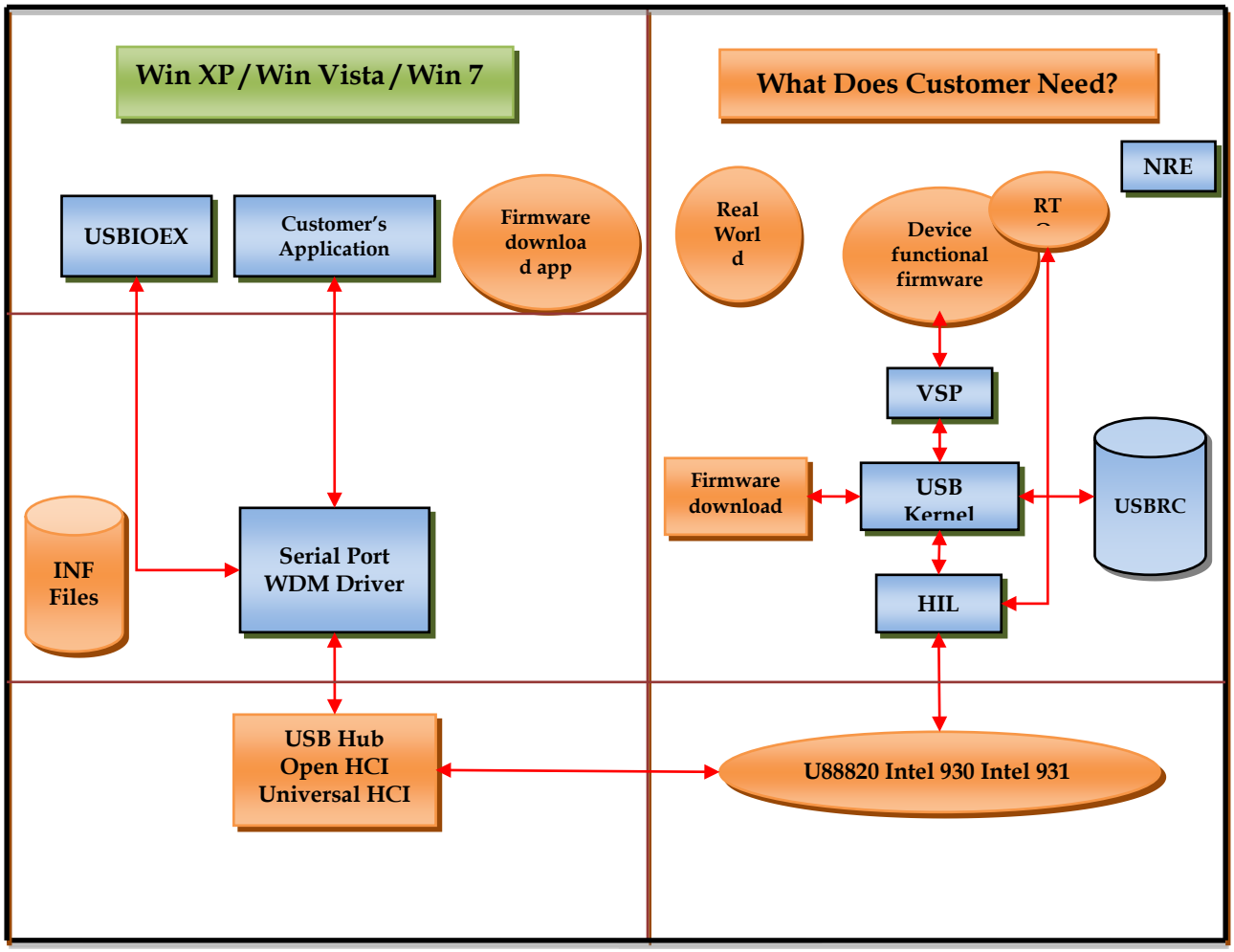


Figure 8 represents the various pieces required for a total USB solution. This document addresses the parts shown on the right hand (device) side: the protocol (VSP in this example), the USB DataPump Kernel, and the Hardware Interface Layer (HIL). The USB Resource Compiler (USBRC) is covered in a separate document.

A USB application can accommodate considerable variety in the hardware operation by having multiple configurations that are dynamically changeable. Part of the USB DataPump's job is to create and deactivate the required paths in order to establish a desired configuration.

5. Implementing A Custom Protocol or Application

It is a straightforward task to add a custom protocol or application (or modifying an existing one) to work in the DataPump software environment., as outlined in the following three simple steps:

1. During configuration of the build environment specify files to be added to the build process (these are the new protocol/applicaton files).

2. Edit these source files.

3. Follow the normal build, execute, debug process and iterate as necessary.

Sections 5.1 and 5.2 provide further information on designing a device and further details on the implementation of a custom protocol.

5.1 Designing a Device with the MCCI USB DataPump

The USB DataPump was created with a particular design process in mind: (also refer to Figure 9)

1. Begin by specifying how the device will appear "on the wire." The following information must be specified:
   - The descriptors.
   - The device class specifications to be followed.
   - Any custom commands or protocols that are to be used.
   - The endpoints that are to be assigned to specific functions.

   Completing this process will provide, the information needed to verify the silicon chosen will perform the desired function. The designer can also create a prototype USB resource file which describes the device.

2. Create the USB resource file (a "URC file".) This file contains, in a high level format, the descriptors that are needed to represent the device to the host. This file therefore describes the endpoints, interface settings, device class, and so forth. It also contains the information needed to create any string descriptors that the OEM wishes to include.

   A complete description of USB resource files, and the USB resource compiler, is available at the MCCI web site, http://www.mcci.com.

3. Once the peripheral has been described, the next step is to use the resource compiler to generate three files:
   - a C file containing all of the descriptors in binary form, as an array of chars. Unicode strings are automatically converted as part of this process.

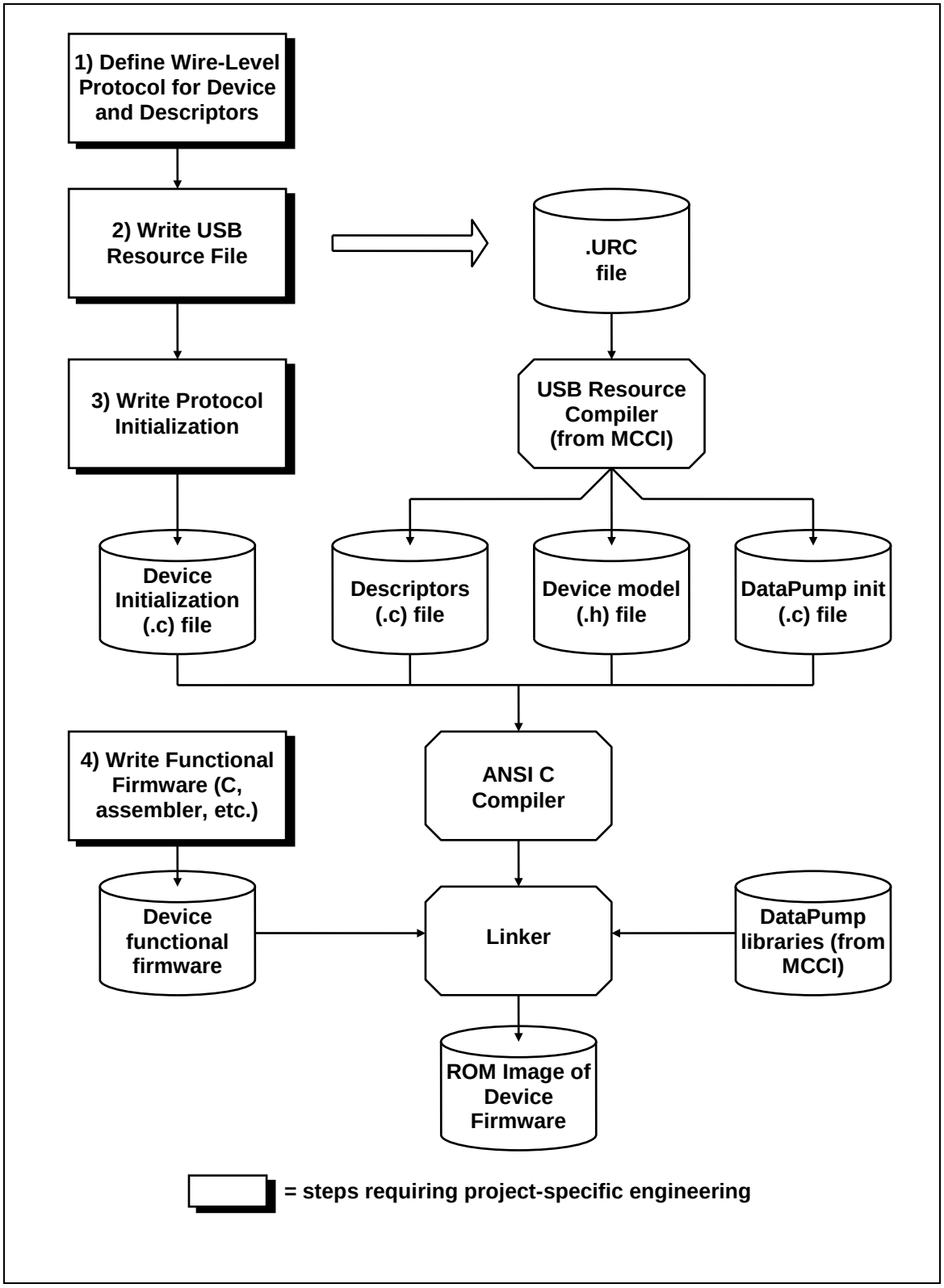
- a C header (.h) file, containing information (number of endpoints, and so forth), and a data structure that models the device. This file is automatically generated and does not need to be edited.
- a C code (.c) file, containing all the code needed to initialize the data structures at runtime to match the description given to the host.

4. A simple initialization function must be created that attaches any protocols needed onto the USB interface.

Upon completion of the process and linking with the MCCI USB DataPump libraries, the MCCI USB DataPump™can automatically support all the USB 1.0/1.1/2.0/3.0 Chapter 9 commands, with no additional programming.

5. Finally, the application code is written that calls the "top edge" of the protocols according to the protocol APIs.

Figure 9. Design Flow



5.2 Implementing a Custom Protocol Using the MCCI USB DataPump

Protocols are implemented as follows:

- The user's initialization function must call the protocol's initialization function.
- The protocol initialization function must allocate and initialize a protocol-specific instance data structure, containing all the information needed to implement that instance of the protocol.
- The protocol initialization function allocates an "event-node", and attaches it to the interface data structure that uses this protocol. For example, if interface 1 is a Communication Class interface, the protocol would attach the event-node to the structure (in the MCCI USB DataPump) that models interface 1.
- The event node contains a pointer to the protocol's event handler.
- When the host configures the device to use that interface, the protocol event handler will be called. The event handler determines that the interface has been activated. If data is to be received from the host, the event handler queues I/O packets to the specific endpoints that can receive data. These packets point to buffers which must be pre-allocated. These packets also contain pointers to completion handlers, which the protocol supplies.
- When the host sends data to the device, the USB DataPump transfers data to the appropriate buffer, and calls the completion routine. The USB DataPump then proceeds to the next packet queued for that endpoint.
- The protocol then gets entered via the completion entry point, does the work required to process the data, and passes it up to the next layer.
- When the device wishes to send data to the host, it simply prepares a packet and queues it to the endpoint. A completion routine will be called to signal that the data has been shipped to the host, so the device can release or reuse the data buffer.
- When the host sends control packets to the interface, the interface event handler again is called, and is allowed to perform whatever protocol-specific handling is required.

Thus, protocols are very modular, and need not duplicate any of the Chapter 9 functionality.

The MCCI USB DataPump software eases the USB programming task by reducing the portions that the customer must write. The core logic remains the same regardless of the application or the processor choice. The low level hardware interface is user adaptable and works with the DataPump through standardized function calls. Likewise, at the application level, other functions make establishing configurations and exchanging data a simpler process.

6. Adding A Custom Hardware Interface

Adding non-USB hardware support, or porting existing C software to work in the DataPump software environment is a straightforward task. Simply follow the following steps:

1. During configuration of the build environment specify files to be added to the build process (these are the new hardware interface specific files).

2. Attach operating system calls and interrupt routines by using the function calls specified in the hardware interface library of the DataPump software (*Section 14 Basic HIL Functions*).

3. Refer to Appendix E Sample Hardware Interface Code for an example of a working hardware interface used as part of the *nicdemo* application.

7. MCCI USB DataPump Data Structures

7.1 Construction of MCCI Derived Type Structures

Most MCCI USB data types are constructed in a manner that allows other types to be derived from them. This allows the types to be extended as needed. Such extensions are typically needed to allow a hardware-specific driver to attach hardware-specific state information to a particular data instance. MCCI prefers this approach to the "device extension" approach because all the data items are contiguous, allowing for simplified MMU support, and allowing various levels of the system to communicate using the same data object. The Derived Data types also allow for stronger type checking in the driver sources because ambiguous pointer types are not used.

MCCI Derived Types start with a base type. This base type represents the highest level of abstraction in the object being modeled. For instance, if a serial port object was being constructed, the base type would represent a generic serial port object. The object would provide a means of getting and sending data to the serial port, as well as a means of listing and controlling the port capabilities.

This serial port object would then be derived to create a type-specific serial port object, say an object for representing a PC standard 16550 port. The new data object would contain all of the elements of the base structure, with the same names as the base structure. It would also contain new elements to track 16550-specific items.

Additional derived objects are then created to describe a 16550 port in a particular hardware application. One derived type might support a 16550 attached directly to CPU I/O pins, while another derived type might describe a 16550 PCMCIA card.

In use, MCCI Derived Types work as if the 'C' language provided means to extend a structure. Each level of deriving preserves the base structure at the front end, while appending the new elements to the back end of the structure. Element names remain constant.

```
struct MCCI_BASETYPE_HDR
   {
   int mbthh_somebaseitem;
   int mbthh_anotherbaseitem;
   };

/*
|| This is the “Embedding Macro”. It is used as the first structure
|| element in any derived structure
*/
#define MBASETYPE_HDR   struct MCCI_BASETYPE_HDR  mbt_hh

struct MBASETYPE
   {
   MBASETYPE_HDR;
   };

/* define sane names */
#define mbt_somebaseitem        mbt_hh.mbthh_somebaseitem
#define mbt_anotherbaseitem     mbt_hh.mbthh_anotherbaseitem

/* A derived type */
struct MDERIVEDTYPE
   {
   MBASETYPE_HDR;
   int mdt_myspecificitem;
   };
```

From the user standpoint, two structures are created; MBASETYPE and MDERIVEDTYPE.

```
struct MBASETYPE
   {
   int mbt_somebaseitem;
   int mbt_anotherbaseitem;
   };

struct MDERIVEDTYPE
   {
   int mbt_somebaseitem;
   int mbt_anotherbaseitem;
   int mdt_myspecificitem;
   };
```

7.2 Basic Types

The following data types may be accessed via the USB DataPump header file, “usbpump.h”.

To minimize recompiles, MCCI uses individual header files for each group of data types. DataPump users may use the simpler "usbpump.h" style, or may include the specific header files identified below:

7.2.1 UBUFQE

UBUFQE is a buffer queue element. It is the main structure used to initiate device data transfers. For each transfer, clients prepare one or more UBUFQEs, and then call UsbPipeQueue() or one of the other APIs from Section 13.1, or other (higher level) APIs that in turn call UsbPipeQueue(). These are all asynchronous APIs, so when processing is done, the DataPump calls a client-supplied callback function. A reentrant code can be conveniently created using the uqe_doneinfo pointer to point to the instance data.

Typedef: UBUFQE, *PUBUFQE

Header file: "ubufqe.h"

```
struct TTUSB_UBUFQE
   {
   UBUFQE_LENGTH   uqe_length;
   UBUFQE_REQUEST  uqe_request;
   UBUFQE      *uqe_next;
   UBUFQE      *uqe_last;
   UBUFIODONEFN *uqe_donefn;
   UPIPE       *uqe_pPipe;
   VOID        *uqe_doneinfo;
   VOID        *uqe_extension;
   VOID        *uqe_buf;
   USBPUMP_BUFFER_HANDLE uqe_hBuffer;
   UBUFQE_USER_EXTENSION uqe_userExtension;
   USHORT      uqe_bufsize;
   USHORT      uqe_bufars;
   USHORT      uqe_bufindex;
   UBUFQE_FLAGS uqe_flags;
   USTAT       uqe_status;
   UCHAR       uqe_timeout;
   };
```

uqe_ length Size of the UBUFQE structure, in bytes. If zero, then legacy rules apply.

uqe_request The kind of operation to be performed by the DataPump for this UBUFQE. If zero, then legacy rules apply

uqe_next This is a forward link the next element in the queue. This should not be modified by client code while the UBUFQE is being processed by the DCD.

uqe_last This is a backward link to the previous element in the queue. This should not be modified by client code while the UBUFQE is being processed by the DCD.

uqe_donefn This is a pointer to function that will be called upon completion of this queue element. The prototype for this function is:

```
VOID (UBUFIODONEFN)(UDEVICE *, UENDPOINT *, UBUFQE *)
```

UDEVICE is a pointer to the current device, UENDPOINT is a pointer to the UENDPOINT used for the request, and UBUFQE is a pointer to the UBUFQE that completed. The client sets this value. The DCD does not change this value. This field must not be NULL.

uqe_pPipe The pipe that this element is for. The user sets this value. The DCD does not change this value.

uqe_doneinfo Context information for completion notification. The user sets this value. The DCD does not change this value.

uqe_extension Extension pointer. This element is private to the DCD. The client may use this field before submitting the UBUFQE to be processed, but the DCD may change this.

uqe_buf Pointer to the data buffer for this operation. The client must set this value. The DCD does not change this value.

uqe_hBuffer The buffer handle associated with uqe_buf. If this value is NULL, the DCD may use this field for temporary storage. If non-NULL, the DCD will not change this value.

uqe_userExtension The user extension field. This is a union with two possible views. First, it may be viewed as a UINT_PTR, using uqe_userExtension.all. Second, it may be viewed as a UINT8, using uqe_userExtension.flags. For backwards compatibility, "ubufqe.h" defines uqe_usrflags to be the same as uqe_userExtension.flags.

uqe_bufsize The size of the associated buffer. The user must set this value.

uqe_bufars The buffer actual record size. Prior to completing a UBUFQE, the DCD sets this value the number of bytes processed from the buffer.

uqe_bufindex The DCD uses this value to track current location in the buffer in any way convenient to the DCD. The value upon completion is unspecified.

uqe_flags Control flags. The client initializes this bit field to provide additional information about the operation to be performed. The DCD may update some of these flags. The names of the various bits are given in Table 8.

uqe_status Completion status. The driver sets this value to inform the user of the completion status. Defined values are listed in Table 7.

uqe_timeout Optional timeout, in frames. Set by user. Only used if UBUFQEFLAG_IDLE_COMPLETES is set, and only used for bulk and interrupt OUT endpoints.

Table 7. uqe_status Values

Symbolic Name	Value	Description
USTAT_BUSY	0	The buffer is in use or queued.
USTAT_OK	1	The buffer completed successfully.
USTAT_KILL	2	The buffer has been canceled.
USTAT_NOTCFG	3	The device is not configured.
USTAT_IOERR	4	A read/write error (ISO) has occurred.
USTAT_REPLACED	5	The packet has been superseded.
USTAT_STALL	6	The endpoint is stalled.
USTAT_ISOBUF	7	Malformed ISO buffer.
USTAT_DEFINITE_- LENGTH_OVERRUN	8	Too much data sent
USTAT_LENGTH_UNDERRUN	9	Not enough data sent!
USTAT_INVALID_PARAM	10	Invalid parameter
USTAT_HCD_CRC	11	Invalid crc
USTAT_HCD_BITSTUFF	12	Bitstuff error
USTAT_HCD_NO_RESPONSE	13	No response
USTAT_HCD_PIDCHECK	14	Bad PID check code
USTAT_HCD_INVALID_PID	15	Invalid or unknown PID
USTAT_DMA_OVERRUN	16	RX DMA too slow
USTAT_DMA_UNDERRUN	17	TX DMA too slow
USTAT_NOHW	18	No HC or DC hardware

Symbolic Name	Value	Description
USTAT_IN_USE	19	Port has pending URBs
USTAT_NO_MEMORY	20	Cannot allocate memory
USTAT_NO_BANDWIDTH	21	No bus bandwidth
USTAT_NO_BUS_POWER	22	Not enough bus power
USTAT_INTERNAL_ERROR	23	Internal error
USTAT_NOT_SUPPORTED	24	Not supported error
USTAT_BABBLE_DETECTED	25	Babble detected
USTAT_BAD_START_FRAME	26	Bad start frame
USTAT_NOT_ACCESSED	27	Not accessed error
USTAT_MISSED_MICRO_FRAME	28	Missed micro-frame
USTAT_ERROR_HANDSHAKE	29	Receive error handshake
USTAT_TRANSACTION_ERROR	30	Transaction error
USTAT_FIFO_ERROR	31	FIFO error
USTAT_ISO_NOT_ACCESSED_BY_HW	32	HC did not access TD
USTAT_ISO_NOT_ACCESSED_LATE	33	Did not submit packet on time
USTAT_TRB_ERROR	34	TRB error
USTAT_NO_RESOURCE	35	No resource available
USTAT_NO_SLOT	36	No slot available
USTAT_INVALID_STREAM_TYPE	37	Invalid stream type
USTAT_SLOT_NOT_ENABLED	38	Slot is in disabled state
USTAT_ENDPOINT_NOT_ENABLED	39	Endpoint is in disabled state
USTAT_BANDWIDTH_OVERRUN	40	Isoch bandwidth overrun

Symbolic Name	Value	Description
USTAT_CONTEXT_STATE_ERRO R	41	Illegal context state
USTAT_NO_PING_RESPONSE	42	Not receive PING_RESPONSE in time
USTAT_MISSED_SERVICE	43	Missed service for isoch
USTAT_ISO_BUFFER_OVERRUN	44	Isoch buffer overrun
USTAT_INVALID_STREAM_ID	45	Invalid stream id
USTAT_SPLIT_TRANSACTION	46	Split transaction error
USTAT_NO_STREAM_SUPPORT	47	Device not support stream

Table 8. uqe_flags Bit Assignments

Symbolic name	Bit #	Description
UBUFQEFLAG_PREBREAK	0	Break stream before. Only applies to data sent to the host. If set, the DataPump will not combine that buffer-full of data with a previous buffer-full to form a packet. Otherwise, for bulk and interrupt endpoints, the DataPump will operate in a streaming mode, combining UBUFQEs of data to form full packets as necessary. This flag is not used in UBUFQEs used for receiving data from the host.
UBUFQEFLAG_POSTBREAK	1	Break stream after. When transmitting data to the host, if this bit is set, the DataPump will ensure that the data from this UBUFQE is not combined with any subsequent buffers, and that effectively this UBUFQE ends the USB transfer. When receiving data from the host, this bit is an output from the DataPump. The DataPump will set this flag if (1) the end of the data in the buffer represents a packet boundary;(2) the inbound packet was a short packet; and (3) the endpoint is a BULK, CONTROL, or INTERRUPT endpoint. Otherwise the DataPump will reset this flag. Note that if uqe_len is a multiple of wMaxPacket size, and the host sends enough bytes to completely fill the buffer, UBUFQEFLAG_POSTBREAK will not be set, even if the next packet on the bus is a zero-length packet. (That ZLP will have to be received by the next UBUFQE.)
UBUFQEFLAG_SYNC	2	Synchronous status. Applies to completion in both directions. When transmitting data to the host, this will cause the buffer

Symbolic name	Bit #	Description
		completion to be delayed until the data has actually been transferred to the host (not just put into the buffer). Not all DCD hardware supports this flag. In that case, the flag is silently ignored.
UBUFQEFLAG_STALL	3	Stall request. To stall an endpoint, create a UBUFQE with this flag set, and with a zero-length buffer. When the UBUFQE reaches the head of the queue for the pipe, the DataPump will halt the endpoint.
UBUFQEFLAG_-SHORTCOMPLETES	4	Short packets complete early. When receiving a packet from the host, the caller should set this flag if a short packet (for bulk/control/interrupt) or missing packet (for iso) is to cause an early completion of the UBUFQE.
UBUFQEFLAG_DEFINITE_-LENGTH	5	Length is definite, and must not cause a split. When receiving data from the host, the caller should set this flag if the caller expects exactly a certain amount of data, and any excess in the last packet should cause an error condition. This flag is often used when receiving setup info from the host. This normally has effect only on the last packet that goes into the buffer.
UBUFQEFLAG_IDLE_-COMPLETES	6	Idle time on the bus completes a partially-filled packet. When receiving data from the host, the caller should set this flag if the timeout value is to be used as an idle time.
UBUFQEFLAG_PROTO_-RESERVED	7	Reserved for use by protocols. As an example in our case, VSP uses this bit. If a class protocol uses UBUFQEs as part of its API, the protocol may reserve this for its own internal use, or it may delegate use to the client.
UBUFQEFLAG_ISOCH_ASAP	8	Isochronous transfer flag for UBUFQE_RQ_ISOCH_IN and UBUFQE_RQ_ISOCH_OUT. If set, the DataPump sends the first packet as soon as possible, and updates IsochStartFrame to the starting frame number. If not set, the client must set IsochStartFrame to the desired frame number for the first packet.
UBUFQEFLAG_INTERNAL_BUFFER_ENTERSCOPE	12	This flag is reserved for internal use by the DCD.
UBUFQEFLAG_INTERNAL_ISOCHDESC_ENTERSCOPE	13	This flag is reserved for internal use by the DCD.

7.2.1.1 Completion Processing

When the DataPump completes processing of a UBUFQE, it calls the client-supplied UBUFIODONEFN given in uqe_donefn.

The UBUFIODONEFN has the following prototype:

```
typedef VOID (UBUFIODONEFN)(
   UDEVICE *pDevice,
   UENDPOINT *pEndpoint,
   UBUFQE *pQe
   );
```

The UBUFIODONEFN must inspect `pQe->uqe_status` to learn the completion status of the UBUFQE, and `pQe->uqe_doneinfo` to fetch any required client context.

The UBUFIODONEFN is always called in DataPump context – so it is never called asynchronously relative to other DataPump operations.

7.2.2 UBUFQE_GENERIC

UBUFQE_GENERIC is a generic UBUFQE that can be used for any request. This is defined for symmetry, but most protocols use the basic UBUFQE instead (both for historical reasons, and to to save memory).

Typedef: UBUFQE_GENERIC, *PUBUFQE_GENERIC

```
union UBUFQE_GENERIC
         {
         struct TTUSB_UBUFQE      UbufqeLegacy;
         UBUFQE_IMPLICIT          UbufqeImplicit;
         UBUFQE_STREAM_IN         UbufqeStreamIn;
         UBUFQE_STREAM_OUT        UbufqeStreamOut;
         UBUFQE_STREAM_INOUT      UbufqeStreamInOut;
         UBUFQE_PACKET_IN         UbufqePacketIn;
         UBUFQE_PACKET_OUT        UbufqePacketOut;
         UBUFQE_PACKET_INOUT      UbufqePacketInOut;
         UBUFQE_ISOCH_IN          UbufqeIsochIn;
         UBUFQE_ISOCH_OUT         UbufqeIsochOut;
         UBUFQE_ISOCH_INOUT       UbufqeIsochInOut;
         UBUFQE_SS_STREAM_IN      UbufqeSsStreamIn;
         UBUFQE_SS_STREAM_OUT     UbufqeSsStreamOut;
         UBUFQE_SS_STREAM_INOUT   UbufqeSsStreamInOut;
         };
```

UbufqeImplicit	Implicit queue-element layout(s)
UbufqeStreamIn	Ubufqe for StreamIn Transfer.
UbufqeStreamOut	Ubufqe for StreamOut Transfer.
UbufqeStreamInOut	Ubufqe for StreamInOut transfer.
UbufqePacketIn	Ubufque for Packet In data transfer.
UbufqePacketOut	Ubufque for Packet Out data transfer.
UbufqePacketInOut	Ubufque for Packet In/Out data transfer.
UbufqeIsochIn	Ubufque for Isochronous In data transfer
UbufqeIsochOut	Ubufque for Isochronous Out data transfer
UbufqeIsochInOut	Ubufque for Isochronous In/Out data transfer
UbufqeSsStreamIn	Ubufque for Super Speed Stream In data transfer
UbufqeSsStreamOut	Ubufque for Super Speed Stream Out data transfer
UbufqeSsStreamInOut	Ubufque for Super Speed Stream In/Out data transfer

7.2.2.1 UBUFQE_TO_GENERIC

This macro provides a reasonably safe way to convert a legacy UBUFQE pointer to a UBUFQE_GENERIC pointer, without using an explicit type cast.

```
UBUFQE_GENERIC *
   UBUFQE_TO_GENERIC(
       UBUFQE *pQe
       );
```

7.3 Isochronous Data Transfers

USB Isochronous transfer types allow endpoints to send (for IN endpoints) or receive (OUT endpoints) a guaranteed number of bytes in each frame or microframe. Isochronous endpoints are allowed when operating at full speed, high speed, and USB 3.1 speeds.[2]

For control, bulk and interrupt pipes, the data flow is divided into USB packets implicitly, according to the wMaxPacketSize associated with the pipe. For isochronous pipes, the DataPump client specifies the packet size for each packet in the buffer. This can range from zero bytes to the maximum packet size, but short packets don't have a special meaning for isochronous transfers. When sending data, the client must specify the size of each packet. When receiving data, the client expects the size of each packet, and the status for each packet (whether it was transferred successfully or not).

The DataPump has two ways of performing isochronous data transfer, "*interleaved isochronous transfers*" and "*descriptor-based isochronous transfers*".

For interleaved transfers, a single data buffer is used, and packet information is combined with ("interleaved") with data (section 7.3.1). For descriptor-based transfers, the client provides two buffers, one for data and the other containing an array of descriptors that identify each packet to be transferred within the data buffer (section 7.3.2).

The client chooses which API to use by preparing different kinds of UBUFQEs.

Unfortunately, not all DCDs support both formats. Generally, MCCI DCDs introduced since 2008 support descriptor format exclusively, both because it is more DMA-friendly and because it is architecturally similar to isochronous support in the MCCI USB host stack (in fact, it uses the same data structures). Older DCDs tend to support interleaved format only. MCCI class protocol implementations typically hide this from the client, but if coding directly to the UBUFQE-based APIs, this must be taken into consideration.

[2] In addition to 5 Gbps at Gen1 and 10 Gbps at Gen2, USB 3.1 also allows for SuperSpeed Inter-chip (SSIC) USB, which has differnet transfer rates.

7.3.1 Interleaved isochronous transfers

For interleaved transfers, the client uses the request code UBUFQE_RQ_PACKET_IN or UBUFQE_RQ_PACKET_OUT, and prepares UBUFQE_PACKET_IN, or a UBUFQE_PACKET_OUT structure.[3]

The uqe_buf pointer and uqe_len size field should be set to point to a buffer consisting of an array of (variable-size) UISOBUFHDR objects, one such object per isoch packet to be sent or received.

7.3.1.1 UISOBUFHDR

When the DataPump receives isochronous data, it builds an interleaved header/data stream in the input buffer. When it transmits isochronous data, it expects to find a similar structure.

When receiving, the DataPump uses a fixed amount of buffer space for each header/data combination, as determined by `UISOBUFPACKET_ROUNDSIZE(wMaxPacketSize)`. Any fractional entries at the end of the buffer are ignored. Even though the UISOBUFPACKET structure is variable length, each entry in a given buffer is the same size. **All header data in the buffer is ignored, and is overwritten**. There is no way to get variable spacing, or to get a different overall size for the individual UISOBUFPACKETs.

When sending, the DataPump processes the buffer as a sequence of consecutive UISOBUFPACKET elements. Unlike when receiving, the DataPump allows the UISOBUFHDRs to be shorter than, or longer than, the wMaxPacketSize. (Of course, the associated data buffer must always be smaller than the wMaxPacketSize for the pipe.)

Typedef: UISOBUFHDR, *PUISOBUFHDR

Header file: #include “uisobufhdr.h”

```
struct TTUSB_ISOBUFHDR
   {
   UINT16  isohdr_size;
   UINT16  isohdr_ars;
   };
```

isohdr_size A word that describes the size of the UISOBUFHDR: the size of the header + size of the data + any padding required to get to the next header. When receiving, this is automatically filled in by the DataPump. When transmitting, this must be prepared by the client. This size should include any required "round-up" so that the next UISOBUFHDR is naturally aligned.

[3] Legacy code may use UBUFQE_RQ_IMPLICIT and UBUFQE_IMPLICIT structures, or may use a bare UBUFQE, which is equivalent.

isohdr_ars The actual record size ("ars") for the data in the buffer, which is the count of the number of bytes actually in the data stream. This must be less than or equal to the size.

7.3.1.2 UISOBUFPACKET

For convenience when actually building or processing an ISO packet, here's a structure containing the header and the data.

Typedef: UISOBUFPACKET, *PUISOBUFPACKET

```
typedef struct TTUSB_ISOBUFPACKET
   {
   UISOBUFHDR isobuf_hdr;

   /* variable-length data [0..isobuf_hdr.isohdr_ars */
   UCHAR   isobuf_data[1];
   };
```

7.3.1.3 UISOBUFPACKET_ROUNDSIZE()

Given the size of an isochronous data packet, UISOBUFPACKET_ROUNDSIZE() calculates how many bytes of buffer-space are needed to hold the packet, a header, and any alignment bytes at the end.

In order to calculate a suitable buffer size for receiving nPackets, calculate `nPackets * UISOBUFPACKET_ROUNDSIZE(wMaxPacketSize)`.

```
#define UISOBUFPACKET_ROUNDSIZE(
       /* UINT16 */ pktsize
       )
```

7.3.2 Descriptor-based isochronous transfers

Descriptor-based transfers differ from interleaved transfers in several important ways:

1. An extended UBUFQE is used, UBUFQE_ISOCH_IN (for transmit) or UBUFQE_ISOCH_OUT (for receive). uqe_request must be set to either UBUFQE_RQ_ISOCH_IN or UBUFQE_ISOCH_OUT. (So you must allocate larger UBUFQEs than you would normally.)
2. The information about the packets is in a second buffer, which is an array of fixed-size USBPUMP_ISOCH_PACKET_DESCR elements. The size (in bytes) of the array is specified by `UBUFQE_ISOCH_INOUT::IsochDescrSize`, and the base is given by `UBUFQE_ISOCH_INOUT::pIsochDescr`.
3. The number of packets transferred is controlled by the number of elements in the array of descriptors.
4. At the end of processing, the DataPump sets `UBUFQE::uqe_bufars` based on the last element processed in the descriptor array. Normally it will be the same as `UBUFQE_ISOCH_INOUT::IsochDescrSize`, but in case of errors (unplug, USB reset, etc.), it will be set to the byte position of the first descriptor that was not processed.

7.3.2.1 UBUFQE_ISOCH_IN, UBUFQE_ISOCH_OUT and UBUFQE_ISOCH_INOUT

Typedef: UBUFQE_ISOCH_IN, UBUFQE_ISOCH_OUT, UBUFQE_ISOCH_INOUT

```
typedef struct
   {
   // the normal UBUFQE contents with VOID *uqe_buf;
   UBUFQE_HDR(VOID *);

   // the additional things for isochronous
   USBPUMP_ISOCH_PACKET_DESCR   *pIsochDescr;
   USBPUMP_BUFFER_HANDLE        hIsochDescr;
   BYTES                        IsochDescrSize;
   UINT32                       IsochStartFrame
   } UBUFQE_ISOCH_IN;
```

`UBUFQE_ISOCH_OUT` and `UBUFQE_ISOCH_INOUT` are the same as `USBPUMP_ISOCH_IN`, except that `UBUFQE_ISOCH_OUT` uses `CONST VOID *uqe_buf`, and `UBUFQE_ISOCH_INOUT` uses `USBPUMP_BUFFER_POINTER uqe_buf`.

pIsochDescr Points to the base of an array of USBPUMP_ISOCH_PACKET_DESCRs.

hIsochDescr Buffer handle for the isochronous descriptor array, or NULL. (If NULL, the DCD may use this field for holding a buffer descriptor that it allocates and frees during the processing of the UBUFQE.)

IsochDescrSize Size of the array of isochronous descriptors, in bytes.

IsochStartFrame If uqe_flags::UBUFQEFLAG_ISOCH_ASAP is set by the client prior to submitting the UBUFQE, this field is an output; the DCD sets it to the **standard** frame number during which the first packet was transferred. Otherwise, the client must initialize this field to the desired starting frame number (again, as a **standard** frame number, not a native fame number).

Standard bus frame numbers are only 11 bits wide. The most-significant bits of this field are ignored by the DCD for input, and set to zero when output.

7.3.2.2 USBPUMP_ISOCH_PACKET_DESCR

This structure describes a single isochronous packet.

Typedef: USBPUMP_ISOCH_PACKET_DESCR *, *PUSBPUMP_ISOCH_PACKET_DESCR

```
typedef struct USBPUMP_ISOCH_PACKET_DESCR
   {
   UINT32          uOffset;
   UINT16          usLength;
   UINT8           ucStatus;
   UINT8           ucSpare;
   };
```

uOffset Specifies the offset of the packet buffer within the overall transfer buffer (uqe_buf). This field is set by the client, and never modified by the DataPump. The values of uOffset must be monotonically increasing within the descriptor array, and uOffset+usLength must be strictly less than or equal to uqe_buflen.

usLength Specifies the size of the data for this packet. For IN pipes, this is the size of the data to be sent, and is not modified. For OUT pipes, the client must initialize this to the maximum desired packet size, and the DataPump will update the value the actual number of bytes received for that packet.

ucStatus The DataPump sets this to the status of each packet transfer: USTAT_OK for success or some other non-zero value for failure.

ucSpare Spare byte for alignment; initialize to zero.

7.3.3 USBPUMP_FRAME_NUMBER

The descriptor-based isochronous APIs need the current frame number. Sometimes that frame number needs to be converted to real time. Variants of USB use different ideas of time. For that reason, the `UsbDeviceGetFrameNumber()` API uses a complex data structure to represent the current bus time.

Typedef: USBPUMP_FRAME_NUMBER, *PUSBPUMP_FRAME_NUMBER

```
typedef struct USBPUMP_FRAME_NUMBER
   {
   UINT32          StandardFrame;
   UINT64          NativeFrame;
   UINT32          Numerator;
   UINT32          Denominator;
   };
```

StandardFrame The "standard" frame count, in milliseconds (always compatible with full speed), as a 32-bit number. Some hardware/DCDs may not be able to economically maintain a long-term frame number; for these, the frame number may be truncated to 11 bits.

NativeFrame The "native" frame count, (microframes for high speed, frames for full speed, other values for technologies like Wireless USB) -- as a 64-bit number.

Numerator, Denominator
Numerator and denominator for converting native frame count to standard frame count. These are set to 1/1 for full speed, 1/8 for high speed, and other values for other technologies. (Native Wireless USB is 1000/1024, simplified to 125/128)

7.4 USB Device Representation

7.4.1 UDEVICESWITCH

The device controller driver (DCD) for the USB device silicon exports thirteen functions to the USB DataPump. These functions are in a table that is (normally) stored in ROM, the UDEVICESWITCH.

Client code must not call these functions directly. Instead, client code must use the UDEV…() macros described in Section 7.4.1.1. Failure to follow this rule will render client code non-portable to future DataPump releases.

Declarations:

```
typedef USTAT UDEVINITIALIZEFN (
   CONST UDEVICESWITCH *,       /* from DCD */
   UDEVICE *,
   BYTES,                       /* memory for the actual structure */
   UPLATFORM *,                 /* from app: the platform structure */
   UHIL_BUSHANDLE,
   IOPORT,                      /* from app: the base address of USB DCI */
   UDEVICE_INITFN *,            /* from USBRC */
   CONST USBRC_ROOTTABLE *,     /* from USBRC */
   UINT32,                      /* from app */
   CONST VOID *,                /* config info */
   BYTES nConfigData            /*  DCD specific. */
   );


typedef VOID UDEVSTARTFN (UDEVICE *);
typedef VOID UDEVSTOPFN (UDEVICE *);
typedef VOID UDEVREPORTEVENTFN (UDEVICE *, ARG_UEVENT, VOID *);
typedef VOID UDEVREPLYFN (
   UDEVICE *,
   CONST VOID *,
   BYTES,           /* lenRequested*/
   BYTES,           /* lenActual*/
   UBUFIODONEFN *,  /*pDoneFn*/
   VOID *,          /*pDoneCtx*/
   UBUFQE_FLAGS     /*flags*/
   );
typedef VOID UDEVREMOTEWAKEUPFN (UDEVICE *);
typedef VOID* UDEVALLOCATEFN (UDEVICE *);
typedef VOID UDEVFREEFN (UDEVICE *);
typedef VOID UDEVGETUDEVICESIZEFN (UDEVICE *);
```

```
typedef VOID UDEVGETUENDPOINTSIZEFN (UDEVICE *);
typedef VOID UDEVENDPOINTINITFN (
   UDEVICE *,   /* pDevice */
   UPIPE *,     /* pPipe */
   UENDPOINT *, /* pEndpoint */
   UCHAR        /* EndpointId */
   );
typedef VOID UDEVGETFRAMENUMBERFN (
   UDEVICE *,   /* pDevice */
   USBPUMP_FRAME_NUMBER *,  /* pFrameNumber */
   );

typedef USBPUMP_IOCTL_RESULT UDEVIOCTLFN (
   UDEVICE *,
   USBPUMP_IOCTL_CODE,
   CONST VOID *,
   VOID *
   );
```

Typedef: UDEVICESWITCH, *PUDEVICESWITCH

```
struct TTUSB_UDEVICESWITCH
   {
   UDEVINITIALIZEFN    *udevsw_Initialize;
   UDEVSTARTFN         *udevsw_Start;
   UDEVSTOPFN          *udevsw_Stop;
   UDEVREPORTEVENTFN   *udevsw_ReportEvent;
   UDEVREPLYFN         *udevsw_Reply;
   UDEVREMOTEWAKEUPFN  *udevsw_RemoteWakeup;
   UDEVALLOCATEFN      *udevsw_AllocateDeviceBuffer;
   UDEVFREEFN          *udevsw_FreeDeviceBuffer;
   UDEVIOCTLFN         *udevsw_Ioctl;
   UDEVGETUDEVICESIZEFN *udevsw_GetUdeviceSize;
   UDEVGETUENDPOINTSIZEFN  *udevsw_GetUendpointSize;
   UDEVENDPOINTINITFN *udevsw_InitUendpoint;
   UDEVGETFRAMENUMBERFN  *udevsw_GetFrameNumber;
   };
```

udevsw_Initialize is a pointer to an exported DCD function that the DataPump calls to initialize the DCD.

udevsw_Start is a pointer to an exported DCD function that the DataPump calls to start up (or restart) the device.

udevsw_Stop is a pointer to an exported DCD function that the DataPump calls to stop the device.

udevsw_ReportEvent	is a pointer to an exported DCD function that the DataPump calls to report device events.
udevsw_Reply	is a pointer to an exported DCD function that the DataPump calls to reply to the current control transaction.
udevsw_RemoteWakeup	is a pointer to an exported DCD function that the DataPump calls to request a remote wakeup.
udevsw_AllocateDeviceBuffer	is a pointer to an exported DCD function that the DataPump calls to allocate a DMA buffer that can be accessed by the device. Not all DCDs require or provide this function.
udevsw_FreeDeviceBuffer	is a pointer to an exported DCD function that the DataPump calls to free the memory space for the device buffer.
udevsw_Ioctl	is a pointer to an exported DCD function that the DataPump calls to access extended IOCTL functions of the DCD.
udevsw_GetUdeviceSize	is a pointer to an exported DCD function that the DataPump calls to get the size of the concrete UDEVICE object.
udevsw_GetUendpointSize	is a pointer to an exported DCD function that the DataPump calls to get the size of the concrete UENDPOINT.
udevsw_InitUendpoint	is a pointer to an exported DCD function that the DataPump calls to initialize an UENDPOINT structure, taking into account any device-specific extensions.
udevsw_GetFrameNumber	is a pointer to an exported DCD function that the DataPump calls to get the current USB bus frame number.

7.4.1.1 Invoking the Device Switch Functions

To initialize:

```
#define UDEVINITIALIZE(
        /*UDEVICESWITCH *           */ pSwitch,
        /* UDEVICE *                */ self,
        /* BYTES                    */ size,
        /* UPLATFORM*               */ platform,
        /* UHIL_BUSHANDLE           */ bushandle,
        /* IOPORT                   */ busport,
```

```
        /* UDEVICE_INITFN *          */ initfn,
        /* CONST USBRC_ROOTTABLE *   */ pRoot,
        /* UINT32 DebugFlags         */ debug,
        /* CONST VOID *              */ configdata,
        /* BYTES                     */ size_config
        )
```

To start the device up:

```
#define UDEVSTART(
        /* UDEVICE *     */ self
        )
```

To stop the device:

```
#define UDEVSTOP(
        /* UDEVICE *     */ self
        )
```

To report a device event:

```
#define UDEVREPORTEVENT(
        /* UDEVICE *     */ self,
        /* ARG_UEVENT    */ event,
        /* VOID *        */ arg
        )
```

To send a reply:

```
#define UDEVREPLY(
        /* UDEVICE *     */ self,
        /* CONST VOID *  */ buf,
        /* BYTES         */ len
        )
```

To request a remote wakeup:

```
#define UDEVREMOTEWAKEUP(
        /* UDEVICE *     */ self
        )
```

To allocate a buffer:

```
#define UDEVALLOCATEFN(
        /* UDEVICE *     */ self
        )
```

To free a buffer:

```
#define UDEVFREEFN(
        /* UDEVICE *     */ self
        )
```

To provide IOCTL functions:

```
#define UDEVIOCTLFN(
        /* UDEVICE *     */ self
        )
```

To get the size of device:

```
#define UDEVGETUDEVICESIZEFN(
        /* UDEVICE *     */ self
        )
```

To get the size of endpoint:

```
#define UDEVGETUENDPOINTSIZEFN (
        /* UDEVICE *     */ self
        )
```

To initialize the endpoint:

```
#define UDEVENDPOINTINITFN (
        /* UDEVICE *     */  pDevice,
        /*UPIPE *        */  pPipe,
        /*UENDPOINT *    */ pEndpoint,
        /* UCHAR         /* EndpointId
        )
```

To get the frame number:

```
#define UDEVGETFRAMENUMBERFN (
        /* UDEVICE *     */  pDevice,
        /* USBPUMP_FRAME_NUMBER *        */  pFrameNumber
        )
```

7.4.1.2 Initializing the Device Switch

```
#define UDEVICESWITCH_INIT_V2(
        /* UDEVREPORTEVENTFN *   */ report,
        /* UDEVREPLYFN *         */ reply,
        /* UDEVSTARTFN *         */ start,
        /* UDEVSTOPFN *          */ stop
        )

#define UDEVICESWITCH_INIT_V3(
        /* UDEVREPORTEVENTFN *   */ report,
        /* UDEVREPLYFN *         */ reply,
        /* UDEVSTARTFN *         */ start,
        /* UDEVSTOPFN *          */ stop,
        /* UDEVREMOTEWAKEUP      */ wakeup
        )

#define UDEVICESWITCH_INIT_V4(
```

```
        /* UDEVINITIALIZEFN *    */ initialize,
        /* UDEVSTARTFN *         */ start,
        /* UDEVSTOPFN *          */ stop,
        /* UDEVREPORTEVENTFN *   */ report,
        /* UDEVREPLYFN *         */ reply,
        /* UDEVREMOTEWAKEUPFN *  */ wakeup,
        /* UDEVALLOCATEFN *      */ allocate,
        /* UDEVFREEFN *          */ free
    )

#define UDEVICESWITCH_INIT_V5(
        /* UDEVINITIALIZEFN *    */ initialize,
        /* UDEVSTARTFN *         */ start,
        /* UDEVSTOPFN *          */ stop,
        /* UDEVREPORTEVENTFN *   */ report,
        /* UDEVREPLYFN *         */ reply,
        /* UDEVREMOTEWAKEUPFN *  */ wakeup,
        /* UDEVALLOCATEFN *      */ allocate,
        /* UDEVFREEFN *          */ free,
        /* UDEVIOCTLFN *         */ ioctl
    )

#define UDEVICESWITCH_INIT_V6(
        /* UDEVINITIALIZEFN *         */ initialize,
        /* UDEVSTARTFN *              */ start,
        /* UDEVSTOPFN *               */ stop,
        /* UDEVREPORTEVENTFN *        */ report,
        /* UDEVREPLYFN *              */ reply,
        /* UDEVREMOTEWAKEUPFN *       */ wakeup,
        /* UDEVALLOCATEFN *           */ allocate,
        /* UDEVFREEFN *               */ free,
        /* UDEVIOCTLFN *              */ ioctl,
        /* UDEVGETUDEVICESIZEFN *     */ getUdeviceSize,
        /* UDEVGETUENDPOINTSIZEFN *   */ getUendpointSize,
        /* UDEVENDPOINTINITFN *       */ uendpointInit
    )

#define UDEVICESWITCH_INIT_V7(
        /* UDEVINITIALIZEFN *         */ initialize,
        /* UDEVSTARTFN *              */ start,
        /* UDEVSTOPFN *               */ stop,
        /* UDEVREPORTEVENTFN *        */ report,
        /* UDEVREPLYFN *              */ reply,
        /* UDEVREMOTEWAKEUPFN *       */ wakeup,
        /* UDEVALLOCATEFN *           */ allocate,
        /* UDEVFREEFN *               */ free,
        /* UDEVIOCTLFN *              */ ioctl,
        /* UDEVGETUDEVICESIZEFN *     */ getUdeviceSize,
        /* UDEVGETUENDPOINTSIZEFN *   */ getUendpointSize,
```

```
            /* UDEVENDPOINTINITFN *      */ uendpointInit,
            /* UDEVGETFRAMENUMBERFN  *   */ getframenumber
        )
```

7.4.2 UDEVICE

The UDEVICE structure represents a single USB device controller to the USB DataPump.

Typedef: UDEVICE, *PUDEVICE

Embedding Macro: UDEVICE_HDR

```
struct TTUSB_UDEVICE
   {
   USBPUMP_OBJECT_HEADER    udev_Header;
   UINT32           udev_ulDebugFlags;
   UINT16           udev_usbPortIndex;
   UINT8            udev_usbDeviceStatus;
   UINT8            udev_usbDeviceAddress;
   UINT32           udev_usbDeviceEnumCounter;
   CONST USBRC_ROOTTABLE *udev_pDescriptorRoot;
   CONST USBIF_DEVDESC_WIRE* udev_pDevDesc;
   USBPUMP_DEVICE_FSM   udev_DeviceFsm;
   UCONFIG          *udev_pAllConfigs;
   UCONFIG          *udev_pConfigs;
   UCONFIG          *udev_pCurrent;
   VOID             *udev_pExtension;
   UPLATFORM        *udev_pPlatform;
   CONST UDEVICESWITCH  *udev_pSwitch;
   UEVENTNODE       *udev_noteq;
   UINTERFACESET    *udev_pCtrlIfcset;
   UENDPOINT        *udev_pEndpoints;
   UINT8            *udev_pReplyBuf;
   UINT8            *udev_pPoolHead;
   UINTERFACESET    *udev_pvAllInterfaceSets;
   UINTERFACE       *udev_pvAllInterfaces;
   UPIPE            *udev_pvAllPipes;
   BYTES            udev_sizeReplyBuf;
   UINT16           udev_wNumAllPipes;
   UINT8            udev_bCurrentSpeed;
   UINT8            udev_bSupportedSpeeds;
   UINT8            udev_RemoteWakeupEnable;
   UINT8            udev_L1RemoteWakeupEnable;
    UDEVICE_LINK_STATE          udev_LinkState;
   UINT8            udev_fSuspendState;
   UINTERFACE *     udev_pFunctionWakeIfc;
```

```
        UINT16          udev_nFunctionWakeIfc;
        UINT16          udev_nAutoRemoteWakeup;
        UINT8           udev_bTestMode;
        UINT8           udev_fLpmEnable;
        UINT8           udev_HnpEnable;
        UINT8           udev_HnpSupport;
        UINT8           udev_AltHnpSupport;
        UINT8           udev_EpActiveState;
        UINT16          udev_wNumAllConfigs;
        UINT8           udev_bNumConfigs;
        UINT8           udev_bNumHSConfigs;
        UINT8           udev_bNumEndpoints;
        UINT8           udev_bNumAllInterfaceSets;
        UINT8           udev_bNumAllInterfaces;
        UINT8           udev_bActiveConfigurationValue;
        UINT16          udev_bmInactiveInEndpoint;
        UINT8           udev_ctlifcset;
        UINTERFACE      udev_ctlifc;
        UINT8           udev_ctlsetupbq;
        UBUFQE          udev_ctlinbq;
        UBUFQE          udev_ctloutbq;
        USETUP_HANDLE   udev_hSetup;
        UINT32          udevhh_fHwFeature_LtmCapable: 1;
        UINT32          udevhh_fValidateEpforAutoRemoteWakeup: 1;
        UINT32          udevhh_fU1Enable: 1;
        UINT32          udevhh_fU2Enable: 1;
        UINT32          udevhh_fLtmEnable: 1;
        UINT32          udev_fDoNotAppendPortIndex: 1;
        USBPUMP_CONTROL_PROCESS_FN   *udev_pControlProcessFn;
        VOID             *udev_pControlProcessCtx;
        };
```

udev_Header The standard USB Object header.

udev_ulDebugFlags The debug flags.

udev_usbPortIndex USB mib port index, support is not complete. Used for implementing the SNMP USB MIB.

udev_usbDeviceStatus USB mib status. Used for implementing the SNMP USB MIB.

udev_usbDeviceAddress Current address. Used for implementing the SNMP USB MIB.

udev_usbDeviceEnumCounter A utility variable, used for implementing the SNMP USB MIB.

udev_pDescriptorRoot	The descriptor root table**.**
udev_pDevDesc	Pointer to the device descriptor for this device instance, from the URC file**.**
udev_DeviceFsm	Device Finite state machine.
udev_pAllConfigs	Pointer to an array of all UCONFIG objects defined for this device**.**
udev_pConfigs	The vector of known configurations for this device.
udev_pCurrent	The current configuration or is set to NULL to indicate that the device is not currently configured.
udev_pExtension	An extension pointer to application-specific data.
udev_pPlatform	The platform for this device.
udev_pSwitch	The switch structure that provides the functional interface for the DataPump to communicate with the HIL for device level interactions.
udev_noteq	The event notification queue for events attached at the device level.
udev_pCtrlIfcset	The default ifcset.
udev_pEndpoints	An array of UENDPOINT structures.
udev_pReplyBuf	The reply buffer.
udev_pPoolHead	The current head of the device pool.
udev_pvAllInterfaceSets	The vector of all interface sets for the device.
udev_pvAllInterfaces	The vector of all interfaces.
udev_pvAllPipes	The vector of all pipes.
udev_sizeReplyBuf	The size (capacity) of the reply buffer in bytes.
udev_wNumAllPipes	The number of pipes, total.
udev_bCurrentSpeed	The current speed (one of USBPUMP_DEVICE_SPEED_LOW, USBPUMP_DEVICE_SPEED_FULL, USBPUMP_DEVICE_SPEED_HIGH,

	USBPUMP_DEVICE_SPEED_WIRELESS, USBPUMP_DEVICE_SPEED_SUPER).
udev_bSupportedSpeeds	Bit mask of supported speeds. Bit 0 for low, bit 1 for full, bit 2 for high, bit 3 for wireless, bit 4 for super.
udev_RemoteWakeupEnable	Boolean. Setting it to TRUE will enable remote wakeup.
udev_L1RemoteWakeupEnable	Boolean. Setting it to TRUE will enable L1 remote wakeup.
udev_LinkState	Controlled by the core DataPump; Set to the selected link power management state.
udev_fSuspendState	Controlled by the core DataPump; set TRUE if suspend state.
udev_pFunctionWakeIfc	Controlled by the core DataPump; save interface pointer which send device notification for function remote wake.
udev_nFunctionWakeIfc	Controlled by the core DataPump; enable counter of interfaces that function remote wake feature is enabled.
udev_nAutoRemoteWakeup	Controlled by the core DataPump; enable counter of automatic remote wakeup feature.
udev_bTestMode	Controlled by the core DataPump; set to the selected test mode (and non-zero) if the device is in test mode.
udev_fLpmEnable	Controlled by the core DataPump; set TRUE when LPM is enabled, FALSE otherwise.
udev_HnpEnable	Controlled by the core DataPump; set TRUE when HNP is enabled, FALSE otherwise.
udev_HnpSupport	Controlled by the core DataPump; set TRUE when HNP is supported by the host controller (as determined by receipt of the OTG HNP support set_feature.
udev_AltHnpSupport	Controlled by the core DataPump; set TRUE when HNP is supported by the host on some ports on the host controller but not necessarily on this one.
udev_EpActiveState	Controlled by the core DataPump; Initialized by device FSM init fn, and read by core DataPump to select proper initial value for EP uep_fActive.
udev_wNumAllConfigs	The size of an array of all UCONFIG objects defined for this device.
udev_bNumConfigs	The number of possible configurations for this device.

udev_bNumHSConfigs The number of high-speed configurations defined in udev_pAllConfigs.

udev_bNumEndpoints The total number of endpoints for this device.

udev_bNumAllInterfaceSets The number of vAllInterfaceSets.

udev_bNumAllInterfaces The number of vAllInterfaces.

udev_bActiveConfigurationValue Set by the core DataPump before report device event; it is configuration value of active configuration.

udev_bmInactiveInEndpoint bitmapped filed identify the inactive IN endpoints.

udev_ctlifcset The control interface.

udev_ctlifc The control interface.

udev_ctlsetupbq The setup buffer queue for the control interface.

udev_ctlinbq The in buffer queue for the control interface.

udev_ctloutbq The out buffer queue for the control interface.

udev_hSetup handle of setup.

udevhh_fHwFeature_LtmCapable Device hardware feature that support LTM or not.

udev_fValidateEpforAutoRemoteWakeup This flag represents we need to validate endpoint for automatic remote wakeup.

udev_fU1Enable This flags represents U1 is enabled or not.

udev_fU2Enable This flags represents U2 is enabled or not.

udev_fLtmEnable This flags represents LTM is enabled or not.

udev_fDoNotAppendPortIndex This flag signals that we don't need to append the USB device port index at the end of the serial number string. This flag will be set to TRUE at the DataPump initialization time if only one USB port is detected.

udev_pControlProcessFn Client registered control packet process function pointer. The function will be registered by USBPUMP_IOCTL_DEVICE_REGISTER_-CONTROL_PROCESS_FN ioctl. This function will

	be called by UsbProcessControlPacket() before starting the common control packet process.
udev_pControlProcessCtx	Context pointer for client-registered control packet process function.

There is one UDEVICE for each USB device interface managed by the DataPump. Think of UDEVICE as representing it's hardware. If writing reentrant client code, UsbPumpObject_GetDevice() can be used to dynamically locate the UDEVICE that governs the specific object. All configuration objects are accessed via the UDEVICE.

One application of multiple UDEVICE instances is for creating USB-to-USB bridges like the MCCI Catena 2210 NCM bridge. Another application is when using both wired USB and MA USB (there might be one UDEVICE for wired USB, and a second UDEVICE for the presentation of the function to MA USB).

With the advent of Type C USB connectors, this architecture will also support multiple concurrent USB device connections for a single platform.

7.4.2.1 Fetching the value of UDEVICE

It is possible to use `UsbPumpObject_EnumerateMatchingNames()` to find all UDEVICEs or to find the specifically wired UDEVICE since its name is predictable. The following sample code shows the procedure:

- First get the DataPump Root object:

```
USBPUMP_OBJECT_ROOT *    CONST pPumpRoot =
                            UsbPumpObject_GetRoot(*pPlatform->upf_Header);
```

- Then for every function (e.g., Modem, Mass Storage, Ethernet) ,

```
USBPUMP_OBJECT_HEADER* pFunctionObject;

pFunctionObject = UsbPumpObject_EnumerateMatchingNames(&pPumpRoot->Header,
                                                       pFunctionObject,
                                                       "storage.*.fn.mcci.com");
```

- And then

```
UDEVICE* CONST pDevice = UsbPumpObject_GetDevice(FunctionObject_p);
```

If "One UDEVICE per USB device hardware instance" is true, in any case, it is easy to use the following sample code to find the device object from the DataPump Root object.

```
USBPUMP_OBJECT_HEADER* pDeviceObject;
pDeviceObject = UsbPumpObject_EnumerateMatchingNames(
   &pPumpRoot->Header,
   NULL,
```

```
    "*.device.mcci.com"
    );
```

Or (more structured)

```
pDeviceObject = UsbPumpObject_EnumerateMatchingNames(
   &pPumpRoot->Header,
   NULL,
   UDEVICE_NAME("*")
   );
```

7.4.3 UCONFIG

This structure represents a single configuration of a USB device.

Typedef: UCONFIG, *PUCONFIG

Embedding Macro: UCONFIG_HDR

```
struct TTUSB_UCONFIG
   {
   UDEVICE          *ucfg_pDevice;
   UINTERFACESET    *ucfg_pInterfaceSets
   UEVENTNODE       *ucfg_noteq;
   VOID             *ucfg_pExtension
   CONST USBIF_CFGDESC_WIRE * ucfg_pCfgDesc
   UINT8            ucfg_Size;
   UINT8            ucfg_bNumInterfaces;
   };
```

ucfg_pDevice The UDEVICE that this configuration belongs to.

ucfg_pInterfaceSets The vector of UINTERFACESET structures for this configuration.

ucfg_noteq The event notification queue pointer for events decorating this configuration.

ucfg_pExtension A pointer to application extension.

ucfg_pCfgDesc A pointer to config descriptor.

ucfg_Size The size of this structure in bytes.

ucfg_bNumInterfaces The number of UINTERFACESET structures supported by this configuration.

7.4.3.1 Accessing the Configuration

To get the size:

```
# define   UCONFIG_SIZE(
        /* UCONFIG *     */ p
        )
```

To set the size:

```
# define   UCONFIG_SETSIZE(
        /* UCONFIG *     */ p,
        /* UINT8         */ size
        )
```

To get the configuration at a specified index:

```
# define   UCONFIG_INDEX(
        /* UCONFIG *     */ newp,
        /* UCONFIG *     */ p,
        /* int           */ index
        )
```

To get the next configuration:

```
# define   UCONFIG_NEXT(
        /* UCONFIG *     */ p
        )
```

7.4.4 UINTERFACESET

This structure represents a collection of interface settings (i.e., the primary interface settings, plus each of the alternatives.)

Typedef: UINTERFACESET, *PUINTERFACESET

Embedding Macro: UINTERFACESET_HDR

```
struct TTUSB_UINTERFACESET
   {
   UCONFIG     *uifcset_pConfig;
   UINTERFACE  *uifcset_pInterfaces;
   UINTERFACE  *uifcset_pCurrent;
   VOID        *uifcset_pExtension;
   UEVENTNODE  *uifcset_noteq;
   UINT8       uifcset_bNumAltSettings;
   UINT8       uifcset_bFlags;
   }
```

uifcset_pConfig The UCONFIG object that owns this interface set.

uifcset_pInterfaces	A vector of pointers to possible alternate interfaces.
uifcset_pCurrent	The currently selected interface.
uifcset_pExtension	An extension pointer to application-specific data.
uifcset_noteq	The event notification queue pointer for events decorating this interface set.
uifcset_bNumAltSettings	The number of alternate settings supported in this interface set. If the interface does not support any alternate settings, i.e., there is only a single setting for the interface, `bNumAltSettings = 1`
uifcset_bFlags	Interface set flag.

7.4.4.1 Accessing the Interface Set

To get the size:

```
# define   UINTERFACESET_SIZE(
        /* UINTERFACESET *   */ p
        )
```

To set the size:

```
# define   UINTERFACESET_SETSIZE(
        /* UINTERFACESET *   */ p,
        /* UINT8             */ size
        )
```

To get the interface set at a specified index:

```
# define   UINTERFACESET_INDEX(
        /* UINTERFACESET *   */ newp,
        /* UINTERFACESET *   */ p,
        /* int               */ index
        )
```

To get the next interface set:

```
# define   UINTERFACESET_NEXT(
        /* UINTERFACESET *   */ p
        )
```

7.4.5 UINTERFACE

This structure represents a single concrete interface (a particular configuration, interface, and alternate interface setting.)

Typedef: UINTERFACE, *PUINTERFACE

Embedding Macro: UINTERFACE_HDR

```
struct TTUSB_UINTERFACE
   {
   UINTERFACESET    *uifc_pInterfaceSet;
   UPIPE            *uifc_pPipes;
   UEVENTNODE       *uifc_noteq;
   VOID             *uifc_pExtension;
   UDATAPLANE       *uifc_pDataPlane;
   UINTERFACE       *uifc _pFunctionIfcNext;
   UINTERFACE       *uifc _pFunctionIfcLast;
   CONST USBIF_IFCDESC_WIRE *uifc_pIfcDesc;
   UINT8            uifc_Size;
   UINT8            uifc_bNumPipes;
   UINT8            uifc_bAlternateSetting;
   UINT8            uifc_bStatus;
   };
```

uifc_pInterfaceSet is a pointer to the UINTERFACESET owning this interface.

uifc_pPipes is a pointer to the first endpoint.

uifc_noteq is the event notification queue for events decorating (targeting to) this interface.

uifc_pExtension is an extension pointer to application-specific data.

uifc_pDataPlane is the linkage to the higher-level (structure agnostic) function drivers.

uifc _pFunctionIfcNext is the linkage to the next interface. Each UINTERFACE can be linked together with other, functionally equivalent UINTERFACEs.

uifc _pFunctionIfcLast is the linkage to the previous interface.

uifc_pIfcDesc is a copy of the config descriptor

uifc_Size is the size of this structure.

uifc_bNumPipes is the number of endpoints in this interface.

uifc_bAlternateSetting is the alternate setting code for this interface.

7.4.5.1 **uifc_bStatus** is the interface status for USB3.Accessing the Interface

To get the size:

```
# define   UINTERFACE_SIZE(
        /* UINTERFACE *  */ p
        )
```

To set the size:

```
# define   UINTERFACE_SETSIZE(
        /* UINTERFACESET *   */ p,
        /* UINT8             */ size
        )
```

To get the interface at a specified index:

```
# define   UINTERFACE_INDEX(
        /* UINTERFACE *  */ newp,
        /* UINTERFACE *  */ p,
        /* int               */ index
        )
```

To get the next interface:

```
# define   UINTERFACE_NEXT(
        /* UINTERFACE *  */ p
        )
```

7.4.6 UPIPE

This structure represents a USB data source or sink. There is one pipe for each valid combination of configuration, interface, alternate interface setting, and endpoint address.

Typedef: UPIPE, *PUPIPE

Embedding Macro: not derivable

```
struct TTUSB_UPIPE
   {
   UINTERFACE  *upipe_pInterface;
   UENDPOINT   *upipe_pEndpoint;
   CONST USBIF_EPDESC_WIRE *upipe_pEpDesc;
   VOID        *upipe_extension;
   UEVENTNODE  *upipe_noteq;
   USHORT      upipe_wMaxPacketSize;
   UCHAR       upipe_bmAttributes;
   UCHAR       upipe_bEndpointAddress;
   };
```

upipe_pInterface is a pointer to the UINTERFACE object that owns this pipe.

upipe_pEndpoint is a pointer to the UENDPOINT object used for this pipe.

upipe_pEpDesc is a pointer to the endpoint descriptor that defines this UPIPE. Set at init-time by USBRC-generated code, and thereafter read-only.

upipe_extension is a pointer to an extension area used for pipe-specific information.

upipe_noteq is the event notification queue for events decorating (targeting to) this interface.

upipe_wMaxPacketSize is the maximum packet size supported on this pipe.

upipe_bmAttributes contains the endpoint type code that specifies the type; CONTROL, ISO, BULK, or INT.

upipe_bEndpointAddress contains the pipe endpoint address.

7.4.7 UENDPOINT

This structure represents each hardware transmit/receive channel. Unlike the pipes, which are associated with configuration setting, interface number, and alternate interface setting, endpoint structures are related to the underlying hardware. Because these structures are used to model the underlying USB interface silicon, endpoints are normally extended by the hardware interface layer to include additional hardware-specific information.

In addition, each endpoint has a pointer to a table of functions that are hardware specific. The USB DataPump talks to the Hardware Interface Layer via this table:

Typedef: UENDPOINT, *PUENDPOINT.

Embedding Macro: UENDPOINT_HDR

```
struct TTUSB_UENDPOINT
   {
   UBUFQE              *uep_pending;
   UPIPE               *uep_pPipe;
   VOID                *uep_pExtension;
   CONST UENDPOINTSWITCH *uep_pSwitch;
   UINT            uep_Size;
   UINT            uep_siolock;
   UINT            uep_stall;
   UINT            uep_fChanged;
   UINT            uep_ucTimeoutFrames;
   UINT            uep_fActive;
   UINT            uep_fActivateWhenComplete;
   };
```

uep_pending The queue of UBUFQE structures being filled.

uep_pPipe The UPIPE that this endpoint is attached to. If this pointer is set to NULL, then no UPIPE is attached and the endpoint cannot do I/O.

uep_pExtension A pointer an optional application-specific extension data area.

uep_pSwitch The switch structure that provides the functional interface for the DataPump to communicate with the DCD (Please see section 7.4.8 UENDPOINTSWITCH) for endpoint level interactions.

uep_Size The size of this structure.

uep_siolock The start-I/O lock-out count.

uep_stall DCD will set to TRUE whenever the endpoint stalls.

uep_fChanged DCD will set to TRUE if configuration of ep changed.

uep_ucTimeoutFrames The timeout down-counter.

uep_fActive Endpoint is active.

uep_ fActivateWhenComplete activate endpoints in the same interface when UBUFQE is completed.

7.4.7.1 Accessing the Endpoint

To get the size:

```
# define   UENDPOINT_SIZE(
        /* UENDPOINT *   */ p
        )
```

To get the endpoint at a specified index:

```
# define   UENDPOINT_INDEX(
        /* UENDPOINT *   */ newp,
        /* UENDPOINT *   */ p,
        /* int           */ index
        )
```

To get the next endpoint:

```
# define   UENDPOINT_NEXT(
        /* UENDPOINT *   */ p
        )
```

To check if a given QE can be put directly as a single packet:

```
# define   UENDPOINT_CANPUTSIMPLE(
        /* UENDPOINT *   */ p,
        /* UBUFQE *      */ pqe,
```

```
        /* BYTES        */ wMaxpacketSize,
        /* BYTES    */ nAvail
        )
```

To calculate the packet size and last UBUFQE for non-simple packets (assumes that UENDPOINT_CANPUTSIMPLE returns False):

```
# define   UENDPOINT_COUNT_PENDING_BYTES(
        /* PUENDPOINT *  */ in ep,
        /* BYTES        */ wMaxpacketSize,
        /* BYTES        */ nAvail,
        /* PUENDPOINT       */ out ep
        )
```

Return the endpoint's eligibility to issue auto remote wakeup. Currently BULK/INT IN pipes are eligible to issue auto remote wakeup.

```
#define UENDPOINT_AUTO_REMOTE_WAKEUP_OK(
        /* CONST UENDPOINT * */ in ep,
        /* BOOL */  fTrueForExamine
        )
```

Use the following macros instead of UENDPOINT_AUTO_REMOTE_WAKEUP_OK() macro:

```
#define UENDPOINT_CAN_AUTO_REMOTE_WAKEUP(
        /* CONST UENDPOINT * */ in ep
        )
#define UENDPOINT_CHECK_AUTO_REMOTE_WAKEUP(
        /* CONST UENDPOINT * */ in ep,
        /* CONST UDEVICE * */    in device
        )
```

7.4.7.2 Initializing the Endpoint

```
# define   UsbGenericEndpointInit(
        /* UENDPOINT *   */ pep
        )
```

7.4.8 UENDPOINTSWITCH

This structure is exported by the DCD. Normally, the DCD supplies a different endpoint switch for each endpoint type supported by the DCD.

Typedef: UENDPOINTSWITCH, *PUENDPOINTSWITCH.

The structure is defined as:

```
struct TTUSB_UENDPOINTSWITCH
   {
   UENDPOINTSWITCH_STARTIO_FN   *uepsw_StartIo;
   UENDPOINTSWITCH_CANCELIOFN   *uepsw_CancelIo;
   UENDPOINTSWITCH_PREPIOFN *uepsw_PrepIo;
   UENDPOINTSWITCH_EVENTFN  *uepsw_EpEvent;
   UENDPOINTSWITCH_TIMEOUTFN    *uepsw_Timeout;
   UENDPOINTSWITCH_QUERYSTATUS_FN   *uepsw_QueryStatus;
   };
```

uepsw_StartIo The DCD function that the DataPump calls to start I/O for this endpoint, after adding one or more UBUFQEs to its pending queue.

uepsw_CancelIo The DCD function that the DataPump calls to cancel all pending UBUFQEs for this endpoint.

uepsw_PrepIo The DCD function that the DataPump calls to prepare a UBUFQE for an operation for this endpoint. This function is optional. If provided, the DataPump calls this for each UBUFQE prior to appending it into the pending queue for the endpoint.

uepsw_EpEvent This DCD function is used by common DCD code to report an event from the device controller hardware to the specific driver for the endpoint.

uepsw_Timeout This DCD function is used by common DCD code for bulk and interrupt OUT endpoints for inactivity timeouts.

uepsw_QueryStatus The DCD function that the DataPump calls to query endpoint status for remote wakeup automation.

7.4.8.1 UENDPOINTSWITCH_STARTIO_FN

```
#include "uendpoint.h"

typedef VOID (UENDPOINTSWITCH_STARTIO_FN)(
               UDEVICE *pDevice,
               UENDPOINT *pEndpoint
               );

typedef UENDPOINTSWITCH_STARTIO_FN USTARTIOFN;
```

```
VOID USTARTIO(
        UDEVICE *pDevice,
        UENDPOINT *pEndpoint
        );
```

Whenever the DataPump has inserted one or more new UBUFQE into the uep_pending list for an endpoint, its calls this function. The DCD then checks whether the endpoint is busy, and if not, scans the pending queue, typically removing as many UBUFQEs as can be passed to the hardware.

(UBUFQEs in the pending queue may be removed by the upper layers at any time, and this might prove embarrassing if DMA or primary ISR activities are still working with the buffer. By moving the UBUFQE to an internal list, the DCD assumes exclusive ownership of the UBUFQE.)

The type USTARTIOFN is for backward compatibility.

The macro USTARTIO() invokes the start-I/O method for the specific endpoint. The preferred style is to use the macro rather than invoking the method function directly.

7.4.8.2 UENDPOINTSWITCH_CANCELIO_FN

```
typedef VOID (UENDPOINTSWITCH_CANCELIO_FN)(
                UDEVICE *pDevice,
                UENDPOINT *pEndpoint,
                USTAT Status
                );

typedef UENDPOINTSWITCH_CANCELIO_FN UCANCELIOFN;

VOID UCANCELIO(
            UDEVICE *pDevice,
            UENDPOINT *pEndpoint
            );
```

Under some circumstances, the DataPump may need to cancel I/O that is pending for a given endpoint. The DataPump can simply remove any UBUFQEs that are in the uqe_pending list, but it must explicitly request the DCD to cancel any UBUFQEs that have been moved to internal lists. It does so by calling this method function.

In response to this primitive, the DCD begins the process of wrapping up any operations that are in-process for the given endpoint. The DCD may delay completions as long as needed for hardware to complete.

Upon return from this call, the portable code can only assume that the UBUFQEs will eventually be completed.

The parameter Status specifies the status to be used when completing the affected UBUFQEs.

The macro UCANCELIO() invokes the cancel I/O method for a given endpoint.

UCANCELIO() is one of the few endpoint-switch methods that is intended to be called directly by clients.

The type UCANCELIOFN is provided for backwards compatibility.

7.4.8.3 UENDPOINTSWITCH_PREPIO_FN

```
typedef VOID (UENDPOINTSWITCH_PREPIO_FN)(
                UDEVICE *pDevice,
                UENDPOINT *pEndpoint,
                UBUFQE *pQe
                );

typedef UENDPOINTSWITCH_PREPIO_FN UPREPIOFN;

VOID UPREPIO(
           UDEVICE *pDevice,
           UENDPOINT *pEndpoint,
           UBUFQE *pQe
           );
```

Some DCDs may need to do some pre-processing on a UBUFQE before it is placed in the uep_pending list. The DataPump always calls this method for a given UBUFQE prior to inserting it in the list. Most DCDs do not need this, and use a default method implementation provided by the DataPump common code.

The operations that a DCD can perform are very limited – the DCD can fill in the uqe_extension field, and can establish any other needed invariants. The DCD cannot complete the UBUFQE, as there is no way to indicate this back to the core DataPump code.

The macro UPREPIO() is used to invoke the method.

The type UPREPIOFN is a synonym for UENDPOINTSWITCH_PREPIO_FN, and is provided for backward compatibility.

7.4.8.4 UENDPOINTSWITCH_EVENT_FN

```
#include "uendpoint.h"

typedef VOID (UENDPOINTSWITCH_EVENT_FN)(
                UDEVICE *pDevice,
                UENDPOINT *pEndpoint
                );

typedef UENDPOINTSWITCH_EVENT_FN UEPEVENTFN;

VOID UEPEVENT(
       UDEVICE *pDevice,
```

```
        UENDPOINT *pEndpoint
        );
```

This endpoint switch method is intended to be called from the device interrupt processor when an event is detected for the underlying endpoint.

The DCD may provide this routine as a convenience to itself; but the DCD is not required to provide it. It is only called (if at all) from code in the DCD. It is not called from DataPump core code, and it must not be called from DataPump client code.

It's defined as part of the portable UENDPOINTSWITCH so that DCDs don't have to provide a DCD-specific parallel set of endpoint-type-specific methods.

This method can be useful for DCDs managing device control interfaces that have a large number of functionally identical endpoints (which is most common).

However, for some USB device control interfaces like the one in SA-1110, it's easier just to do the work directly in the primary ISR, without this extra level of indirection. Those DCDs don't provide or use this method.

The macro UEPEVENT() invokes this method.

The type UEPEEVNTFN is a synonym for UENDPOINTSWITCH_EVENT_FN, and is provided for backward compatibility.

7.4.8.5 UENDPOINTSWITCH_TIMEOUT_FN

```
#include "uendpoint.h"

typedef VOID (UENDPOINT_SWITCH_TIMEOUT_FN)(
                UDEVICE *pDevice,
                UENDPOINT *pEndpoint
                );

typedef UENDPOINT_SWITCH_TIMEOUT_FN UTIMEOUTFN;

VOID UTIMEOUT(
        UDEVICE *pDevice,
        UENDPOINT *pEndpoint
        );
```

This endpoint switch method is intended to be called by the DCD when an inactivity timeout is detected for the underlying endpoint.

The DCD may provide this routine as a convenience to itself; the DCD is not a convenience to provide it. It is only called (if at all) from code in the DCD. It is not called by DataPump core code, and it is not to be called by DataPump client code.

It is defined as part of the portable UENDPOINTSWITCH so that DCDs don't have to provide a DCD-specific parallel set of endpoint-type-specific methods.

This method can be useful for DCDs managing device control interfaces that have a large number of functionally identical endpoints (which is most common).

However, for some USB device control interfaces like the one in SA-1110, it is easier just to do the work directly from the SOF processing logic, without this extra level of indirection. Those DCDs don't provide or use this method.

The macro UTIMEOUT() invokes this method.

The type UTIMEOUTFN is a synonym for UENDPOINTSWITCH_TIMEOUT_FN, and is provided for backward compatibility.

7.4.8.6 UENDPOINTSWITCH_QUERYSTATUS_FN

```
typedef BOOL (UENDPOINTSWITCH_QUERYSTATUS_FN)(
                    UDEVICE *pDevice,
                    UENDPOINT *pEndpoint
                    );

BOOL UENDPOINTSWITCH_QUERYSTATUS(
            UDEVICE *pDevice,
            UENDPOINT *pEndpoint
            );
```

The DataPump core calls this method in order to determine whether an endpoint is active.

This method need only be implemented for IN endpoints. The default pipe is not required to provide this method for its control-IN endpoint. UsbPumpEndpointSwitch_QueryStatusStub() is a suitable default method for endpoints that don't need this method.

If the endpoint is an IN endpoint, the DCD is required to check its internal queues, and return TRUE if any UBUFQE is pending for this endpoint in an internal queue. (The portable queue has already checked pEndpoint->uqe_pending, so the DCD should not bother with this.)

For IN endpoints, if no UBUFQE is pending, the DCD is also required to check IN FIFOs to see whether traffic is pending to the host (presumably sent by a UBUFQE that already completed). If so, the DCD shall return TRUE.

In all other cases, this method shall return FALSE.

The macro UENDPOINTSWITCH_QUERYSTATUS() is used to invoke this method.

7.5 Events

The USB DataPump allows applications to decorate the USB data structure graph with functions to be called upon the occurrence of events defined by the USB DataPump. Events include such things as 'configure change' (applied to the UCONFIG node for active/inactive transactions), 'interface change' (applied to the UINTERFACE nodes for active/inactive transactions), and set/clear feature, applied to the appropriate node (device/interface/pipe.)

7.5.1 UEVENT

The UEVENT is the base type for holding event codes.

7.5.2 UEVENTFN

The UEVENTFN type is used to describe callback functions used with events. It has the following form:

Typedef: UEVENTFN, *PUEVENTFN.

```
/*
|| use UEVENTFN to declare function prototypes, and PUEVENTFN to
|| declare pointers to functions of type UEVENTFN.
*/
typedef VOID (UEVENTFN)(
                UDEVICE     *pDevice,
                UEVENTNODE  *pEventNode,
                UEVENT      event,
                VOID        *evinfo
                };
```

The fields `pDevice` and `pEventNode` are pointers to the UDEVICE and UENDPOINT nodes for this event. `event` contains a valid event code, and `evinfo` varies based on the value of `event`.

Table 9. Defined UEVENT Codes

Symbolic Name	Value	evinfo->	Description
UEVENT_CONFIG_SET	0	setup packet	Configuration change – passed to new configuration.
UEVENT_CONFIG_UNSET	1	setup packet	Configuration change – passed to old configuration.
UEVENT_IFC_SET	2	setup packet	Interface change – posted to old interface
UEVENT_IFC_UNSET	3	setup packet	Interface change – posted to new interface

Symbolic Name	Value	evinfo->	Description
UEVENT_FEATURE	4	event packet	Set/clear feature – applies to devices, interfaces, and endpoints.
UEVENT_CONTROL	5	UEVENTSETUP	Control packet outcall. Used for vendor-specific packets.
UEVENT_SUSPEND	6	Null	Enter suspend state – posted to device
UEVENT_RESUME	7	Null	Leave suspend state – posted to device
UEVENT_RESET	8	Null	USB reset received - posted to device
UEVENT_SETADDR	9	setup packet	Set address received
UEVENT_CONTROL_PRE	10	UEVENTSETUP	Prescan to test for control packet.
UEVENT_INTLOAD	11	UINTSTRUCT	An interrupt-load event has occurred. Only applies to interrupt event queues.
UEVENT_GETDEVSTATUS	12	buffer	Get device status: arg points to buffer to be filled in. Byte 0 gets the power bit for self/bus powered. Depending on state, byte 1 is reserved. Byte 0 bit 0 is already set to remote wakeup status from portable data base.
UEVENT_GETIFCSTATUS	13	buffer	Get interface status: arg points to buffer to be filled in, 2 bytes long.
UEVENT_GETEPSTATUS	14	buffer	Get endpoint status: arg points to buffer to be filled in.
UEVENT_SETADDR_EXEC	15	setup packet	Execute a set-addr command.
UEVENT_DATAPLANE	16	UEVENTDATAPLANE_INFO	special nested event for dataplanes.
UEVENT_ATTACH	17	Null	bus attach event (not always possible)
UEVENT_DETACH	18	Null	bus detach event (not always possible)

Symbolic Name	Value	evinfo->	Description
UEVENT_PLATFORM_EXTENSION	19	UEVENTPLATFORMEXTENSION_INFO	a platform-specific extension -- this is for use by platfrom-specific code.
UEVENT_L1_SLEEP	20	UEVENT_L1_SLEEP_INFO	enter the sleep (L1) state
UEVENT_CABLE	21	Null	cable detected (device)
UEVENT_NOCABLE	22	Null	cable missed (device)
UEVENT_DETECT_ENDPOINT_ACTIVITY	23	Null	post to DCD to detect endpoint activity
UEVENT_VENDOR_CONTROL	24	UEVENTSETUP	control packet outcall; used for vendor-specific packets.
UEVENT_VENDOR_CONTROL_PRE	25	UEVENTSETUP	pre-scan to test for vendor control packet.
UEVENT_SET_SEL	26	UEVENT_SET_SEL_INFO	post to DCD to set SEL & PEL. (USB3)
UEVENT_SET_ISOCH_DELAY	27	UEVENT_SET_ISOCH_DELAY_INFO	post to DCD to set isochronous delay.
UEVENT_FUNCTION_SUSPEND	28	UEVENT_FUNCTION_SUSPEND_INFO	post to interface to notify function suspend event.
UEVENT_FUNCTION_RESUME	29	Null	post to interface to notify function
UEVENT_FUNCTION_REMOTE_WAKE_CAPABLE	30	UEVENT_FUNCTION_REMOTE_WAKE_CAPABLE_INFO	post to interface to get function remote wake capable.
UEVENT_DEVICE_NOTIFICATION	31	UEVENT_DEVICE_NOTIFICATION_INFO	post to DCD to send device notification.
UEVENT_U1_SLEEP	32	Null	enter the sleep (U1) state. Post to the device notification event queue.
UEVENT_U2_SLEEP	33	Null	Enter the sleep (U2) state.
UEVENT_EXIT_U1_U2	34	Null	post to the DCD to exit U1 or U2 sleep state.

7.5.3 UEVENTNODE

The USB DataPump uses the UEVENTNODE structure to decorate the USB data structure graph with functions to be called upon the occurrence of events defined by the USB DataPump. Events include such things as 'configure change' (applied to the UCONFIG node for active/inactive transactions), 'interface change' (applied to the UINTERFACE nodes for active/inactive transactions), and set/clear feature, applied to the appropriate node (device/interface/pipe.)

Typedef: UEVENTNODE, *PUEVENTNODE.

Embedding Macro: UEVENTNODE_HDR

```
struct UEVENTNODE
   {
   UEVENTNODE  *uev_next;
   UEVENTNODE  *uev_last;
   UEVENTFN    *uev_pfn;
   VOID        *uev_ctx;
   };
```

uev_next Is a forward link pointer to the next event node in the queue.

uev_last Is a backward link pointer to the previous event node.

uev_pfn Is the callback function for this event node.

uev_ctx Is the context pointer for use by the callback function.

The following example uses events structure and method to retrieve configuration value selected by Host:

First prepare a UEVENTNODE in allocated or static memory:

```
UEVENTNODE MyEventNode;
```

Declare an event handler:

```
UEVENTFN MyDeviceEventFunction;
```

Then register it with the UDEVICE using the following:

```
UsbAddEventNode(&pDevice->udev_noteq, &MyEventNode, MyDeviceEventFunction,
pMyContent);
```

The content of MyDeviceEventFunction is:

```
VOID MyDeviceEventHandler(
   UDEVICE *   pDevice,
   UEVENTNODE *pThis Node,
   UEVENT      why,
   VOID *      pEventSpecificInfo
   )
```

```
{
if (why == UEVENT_CONFIG_SET)
   {
   UINT8 * CONST pSetup = pEventSpecificInfo;
   UINT8 * CONST bValue = pSetup[2];
   /* the selected configuration is bValue */
   }
}
```

7.5.4 UEVENTFEATURE

UEVENTFEATURE is used to pass SET/CLEAR FEATURE requests to the appropriate event queue. Several of the fields are unpacked representations of fields given for Device Requests in Chapter 9 of the USB Core Specification.

Typedef: UEVENTFEATURE, *PUEVENTFEATURE.

```
struct UEVENTFEATURE
   {
   USHORT  uef_feature;
   USHORT  uef_index;
   USHORT  uef_value;
   UINT8   *uef_setup;
   union {
           UDEVICE         *pDevice;
           UINTERFACESET   *pInterfaceSet;
           UINTERFACE      *pInterface;
           UPIPE           *pPipe;
           UENDPOINT       *pEndpoint;
         } uef_relstruct;
   };
```

uef_feature Is the feature selector. This corresponds to the wValue element of a device request packet.

uef_index Is the feature index. It corresponds to the wIndex element of a device request packet.

uef_value Is boolean. TRUE indicates a SET request, FALSE indicates a CLEAR request.

uef_setup Is the raw setup packet.

uef_relstruct Is the 'relevant' structure. This union contains a pointer type for each of the base types used to represent a USB device.

7.5.5 USETUP

The USETUP structure corresponds to the USB Device Request Packet structure given in Section 9.3 of the USB core specification. Raw setup packets are unpacked into this structure.

Typedef: USETUP, *PUSETUP.

```
struct USETUP
   {
   UCHAR   uc_bmRequestType;
   UCHAR   uc_bRequest;
   USHORT  uc_wValue;
   USHORT  uc_wIndex;
   USHORT  uc_wLength;
   };
```

uc_bmRequestType corresponds to the bmRequestType field listed in the USB core spec.

uc_bRequest corresponds to the bRequest field listed in the USB core spec.

uc_wValue corresponds to the wValue field listed in the USB core spec.

uc_wIndex corresponds to the wIndex field listed in the USB core spec.

uc_wLength corresponds to the wLength field listed in the USB core spec.

7.5.6 UEVENTSETUP

The UEVENTSETUP type is used to carry data to the appropriate event queue. A raw setup packet is unpacked into the uec_ucp structure and the type is then passed to the appropriate layer based on the values of uec_ucp.uc_bmRequestType and uec_ucp.uc_bRequest.

Typedef: UEVENTSETUP, *PUEVENTSETUP.

```
struct UEVENTSETUP
   {
   UCHAR   uec_accept;
   UCHAR   uec_reject;
   USETUP  uec_ucp;
   };
```

uec_accept, uec_reject Are booleans. The event processing routines are required to set uec_accept to TRUE to accept; and to set uec_reject to TRUE to reject. Both start out as FALSE; and the event processing routines should set the accept field to accept, set the reject field to reject; or else leave both fields alone, so the accept field becomes the logical OR of all the accepts, and the reject field becomes the logical OR of all the rejects.

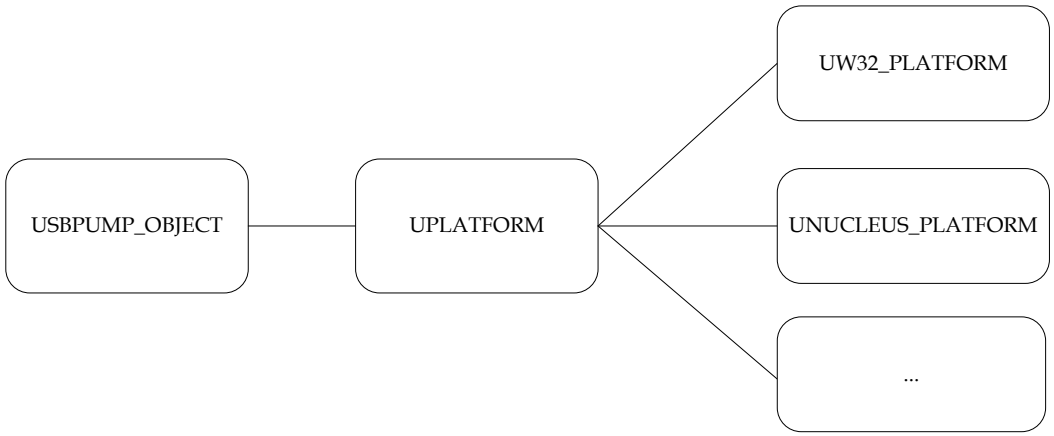
uec_ucp Is the unpacked version of the setup packet data.

7.6 Platform

MCCI Platform represents the underlying operating system and target board to the DataPump. Every UPLATFORM is a DataPump object. Normally, there is only one per DataPump task and the concrete instance for a given platform is derived from UPLATFORM.

7.6.1 UPLATFORM Type Derivation Diagram

Figure 10. Platform Type Derivation



7.6.2 Structure of UPLATFORM

Typedef: UPLATFORM, *PUPLATFORM.

```
struct TTUSB_PLATFORM
   {
   USBPUMP_OBJECT_HEADER    upf_Header;
   USBPUMP_OBJECT_HEADER ** upf_ppHashTbl;
   BYTES                    upf_nHashTbl;
   PUEVENTCONTEXT           upf_pEventctx;
   PUPOLLCONTEXT            upf_pPollctx;
   VOID                     *upf_pContext;
   UPLATFORM_MALLOC_FN      *upf_pMalloc;
   UPLATFORM_FREE_FN        *upf_pFree;
   CONST UHIL_INTERRUPT_SYSTEM_INTERFACE    *upf_pInterruptSystem;
   UPLATFORM_POST_EVENT_FN  *upf_pPostEvent;
   UPLATFORM_GET_EVENT_FN   *upf_pGetEvent;
   UPLATFORM_CHECK_EVENT_FN *upf_pCheckEvent;
```

```
UPLATFORM_YIELD_FN   *upf_pYield;
UPLATFORM_DEBUG_PRINT_CONTROL *upf_pDebugPrintControl;
UPLATFORM_CLOSE_FN   *upf_pPlatformClose;
UPLATFORM_IOCTL_FN   *upf_pIoctl;
UPLATFORM_DI_FN      *upf_pDi;
UPLATFORM_SETPSW_FN  *upf_pSetPsw;
UPLATFORM_CREATE_ABSTRACT_POOL_FN    *upf_pCreateAbstractPool;
UTASK_ROOT       *upf_pTaskRoot;
CONST USBPUMP_TIMER_SWITCH *upf_pTimerSwitch;
VOID         *upf_pTimerContext;
USBPUMP_ALLOCATION_TRACKING  *upf_pAllocationTracking;
USBPUMP_SESSION_HANDLE   upf_hUhilAux;
USBPUMP_UHILAUX_INCALL   *upf_ pUhilAuxIncall;
USBPUMP_ABSTRACT_POOL    *upf_pAbstractPool;
BYTES                    upf_PoolUsed;
UPLATFORM_ABSTRACT_POOL  *upf_pPlatformAbstractPoolHead;
};
```

upf_pHeader DataPump Object header.

upf_ppHashTbl The object header hash table is used to quickly find an object from an object handle.

upf_pEventctx The event context block is used for event processing. It's only dereferenced by the event processing methods, and is treated as a private handle by the rest of the datapump code.

upf_pPollctx The polling context block is used, on a somewhat application-specific basis, to hold context for outcalls for "polling" other subsystems. The sample event loop, for example, uses this. This should be treated as a private handle by modules outside of the polling code (UHIL_DoPoll). In addition, this is obsolescent, and should be superseded by functionality in UPLATFORM_YIELD_FN.

upf_pContext The platform context pointer is a generic, opaque pointer for use by the platform code.

upf_pPostEvent, upf_pGetEvent, upf_pCheckEvent, upf_pEventctx
These pointers provide the abstract DataPump Event API. upf_pPostEvent, upf_GetEvent, and upf_pCheckEvent are function pointers for the methods of the Event API; they must not be NULL. upf_pCheckEvent serves to provide the self pointer to the implementation of the Event API.

upf_pInterruptSystem, upf_pDi, upf_pSetPsw
This pointers provide the abstract interrupt system. upf_pInterruptSystem points to the table of dispatch functions . upf_pDi and upf_pSetPsw are function pointers. These pointers must not be NULL.

upf_pMalloc, upf_pFree

Allocate and free memory functions.

upf_pPlatformIoctl This is optionally provided to do filtering for platform-specific IOCTLs.

upf_pPollCtx, upf_pYield

These optional interface provide some additional control in non-preemptive systems

upf_pDebugPrintControl

This is an optional interface to print debug message.

upf_pCreateAbstractPool

This optional interface creates abstract memory pool.

upf_pTimerSwitch, upf_pTimerContext

This optional interface is used only by the host and OTG stacks; it provides a millisecond timer service.

upf_pTaskRoot This optional interface provides inter-task communication.

upf_pAllocationTracking

This optional interface tracks the amount of dynamically allocated memory required for a given module or configuration of the DataPump.

upf_hUhilAux, upf_pUhilAuxIncall

This mandatory interface provides the buffer handler to HCD request.

upf_pAbstractPool, upf_PoolUsed, upf_pPlatformAbstractPoolHead

These interface provides platform abstract pool information.

7.7 Data Planes and Data Streams

UPIPEs are a convenient abstraction when implementing simple devices, but when implementing more advanced devices, they are too simple, because of the one-to-one correspondence between USB endpoint descriptors and UPIPEs. For example, when implementing a high-speed mass storage device, the user must manage an IN endpoint and an OUT endpoint. A high-speed device is also required to support full-speed, so there are two endpoint descriptors for each logical endpoint; one for high speed and one for full speed. Generally speaking, any time a device has multiple operating modes with different endpoint descriptors, there will be one UPIPE for each endpoint in each operating mode.

Because this is a common problem, the DataPump provides a common solution: the UDATASTREAM/UDATAPLANE abstraction. A UDATASTREAM represents a single logical endpoint across all operating modes. (In effect, all the related UPIPEs are represented by a single UDATASTREAM.)

A UDATAPLANE represents a collection of interface settings, possibly collected from multiple alternate settings, configurations and speeds. In such a collection, only one such setting may be active at a given time.

A UDATAPLANE provides a uniform means of managing this kind of complexity for protocols and device implementations. Each interface may be associated with at most one UDATAPLANE; the DataPump core takes care of delivering any interface level messages to the UDATAPLANE as well.

A UDATAPLANE may be used for control plane management (in cases such as Communication Device Class, where the control plane is associated with a separate interface).

A UDATAPLANE also contains pointers to UDATASTREAMs which are created by the client protocol to represent individual logical data streams.

In operation, UDATASTREAMs are very much like UPIPEs, in that the client submits UBUFQEs to a given UDATASTREAM. The UDATASTREAM determines which UPIPE is currently appropriate, and routes the UBUFQE to that UPIPE. (To continue the mass storage example, a the client code queues a UBUFQE to the OUT UDATASTREAM in order to receive a mass storage command block. For a full-speed only device, the UDATASTREAM simply routes to the underlying UIPE. For a high-speed device with full-speed fall back, the UDATASTREAM routes to the UPIPE for high-speed or full-speed OUT, as determined by the current operating mode of the device.) All the routing policies are set during device initialization; although the client code must supply routine policy at start-up, while the device is operating, the client code doesn't need to worry about device operating speed.

7.7.1 UDATAPLANE

Typedef: UDATAPLANE, *PUDATAPLANE

```
#include "udataplane.h"

struct __TMS_UDATAPLANE
   {
   USBPUMP_OBJECT_HEADER    Header;
   UDATAPLANE           *pNext;
   UDATAPLANE           *pLast;
   CONST UDATAPLANE_OUTSWITCH   *pOutSwitch;
   VOID                 *pClientContext;
   UDEVICE              *pDevice;
   UINTERFACE           *pInterfaceListHead;
   UINTERFACE           *pCurrentInterface;
   UINT32               ulGenerationCount;
```

```
UINT32              ulSavedGenerationCount;
UDATASTREAM         *pDataStreamHead;
UEVENTNODE          *pEventNode;
UINT                fDataPlaneSuspend: 1;
UINT                fRemoteWakeupEnable: 1;
UEVENTNODE          ManagementEventNode;
UEVENTNODE          DeviceEventNode;
};
```

Header A standard USB DataPump object header. By making the UDATAPLANE a named object, we make it discoverable and controllable by loosely-coupled clients.

pNext, pLast Forward and back links. Normally a UDATAPLANE is part of a larger collection.

pOutSwitch A pointer to the outswitch. The client can set this to point to a table of functions that are called back for additional processing. It is usually used by the UDATASTREAM layer.

pClientContext
A pointer to the context which is used by the owner of the UDATAPLANE _OUTSWITCH for communicating context upstream.

pDevice A pointer to the parent device. This is saved for efficiency.

pCurrentInterface
A pointer to the unique interface that is active for this UDATAPLANE. If it is NULL, no such interface is active, which means that none of the UDATASTREAMs associated with this UDATAPLANE can be used to transfer data.

pInterfaceListHead
A pointer to the head of the circularly linked list of interfaces. Each interface is linked using the uifc_pFunctionIfcNext and uifc_pFunctionIfcLast fields. The interfaces are linked in order of discovery. A circular doubly linked list is used for consistency with the rest of the DataPump code.

pEventNode A pointer to the head of the event node chain for this UDATAPLANE. Events affecting any of the underlying interfaces will be broadcast to this chain. This is primarily for internal use; clients should be able to get all info in a more useful form via the UDATAPLANE _OUTSWITCH.

pDataStreamHead
A pointer to the head of the list of UDATASTREAMs associated with this UDATAPLANE.

ulGenerationCount
The generation count can be used by clients to simplify synchronization with the Data Plane / Data Stream mechanism. The count is incremented by the core DataPump whenever a bus event causes the DataPump to begin changing the state of a UDATAPLANE.

ulSavedGenerationCount

This count is maintained by the Data Plane implementation. It is set to a copy of ulGenerationCount whenever the DataPump finishes processing an interface up or down event. Whenever ulGenerationCount is not equal to ulSavedGenerationCount, UDATAPLANE clients can assume that it is probably not a good time to send UBUFQEs towards the host, because the data-structures are in transition.

ManagementEventNode

An eventnode for internal use by the UDATAPLANE implementation.

DeviceEventNode

An eventnode for internal use by the UDATAPLANE implementation.

fDataPlaneSuspend

Flag for UDATAPLANE Suspend.

fRemoteWakeupEnable

Flag for RemoteWakeupEnable.

7.7.2 UDATAPLANE APIs

UDATAPLANE has two major APIs. The UDATAPLANE has a native API, used for most operations. It also provides the standard DataPump Object API.

7.7.2.1 UDATAPLANE Native APIs

These functions are defined in "udataplane.h":

- UsbPumpDataPlane_Create()

Allocate and initialize a UDATAPLANE object.

- UsbPumpDataPlane_Init()

Initialize an existing, uninitialized UDATAPLANE object.

- UsbPumpDataPlane_AddInterface

Add a single interface to a UDATAPLANE.

- UsbPumpDataPlane_AddInterfacesFromSet()

Add all the unclaimed interfaces from an interface set to a UDATAPLANE, creating uDATASTREAMs as appropriate.

- UsbPumpDataPlane_MatchConfig ()

Match data plane with active configuration.

- UsbPumpDataPlaneI_DisconnectStreams ()

Internal method of UDATAPLANE objects: disconnects data streams.

- UsbPumpDataPlane_Deinit()

Internal deinitialize a UDATAPLANE object.

- UsbPumpDataPlane_ReconnectStreams ()

Internal method of UDATAPLANE objects: reconnects data streams.

- UsbPumpDataPlane_ActivateEndpoints()

Activate or deactivate all endpoints associated with a UDATAPLANE.

7.7.2.2 UDATAPLANE Object Interface

Each UDATAPLANE is an MCCI DataPump object. All UDATAPLANES are named "dataplane.obj.mcci.com". The tag is ('U', 'D', 'p', 'l'). There are no UDATAPLANE-specifc IOCTLs, but IOTCLs sent to a given UDATAPLANE instance will be routed to the UDEVICE, UPLATFORM and USBPUMPOBJECT_ROOT objects. It's class parent is inherited from the class parent of UDEVICE.

7.7.3 UDATASTREAM

The UDATASTREAM represents a single logical data stream within a DataPump. It represents a single logical UPIPE. The particular UPIPE represented may change over time, based on the current operating speed of the device, the current configuration of the device, and the current alternate setting of the relevant USB interface.

Typedef: UDATASTREAM, *PUDATASTREAM

```
#include "udatastream.h"

struct __TMS_UDATASTREAM
   {
   UDATASTREAM *pNext;
   UDATASTREAM *pLast;
   UPIPE       *pCurrentPipe;
```

```
UBUFQE      *pHoldQueue;
UDATAPLANE  *pDataPlane;
UCHAR       ucBindingFlags;
UCHAR       ucPipeOrdinal;
USHORT      usEpAddrMask;
};
```

pNext, pLast Pointers to sibling UDATASTREAMs that are controlled by the parent UDATAPLANE.

pCurrentPipe A pointer to the pipe that is currently associated with this data stream. It is set to NULL if no pipe is active.

ucBindingFlags These flags control how the UDATASTREAM library searches for a matching pipe. They are set up when the UDATASTREAM is bound to the UDATAPLANE. They are specified by the client, and indicate what kinds of pipe are acceptable. Each bit, if set, allows a UPIPE of the specified kind to match.

The defined flags are:

UPIPE_SETTING_MASK_CONTROL_IN
UPIPE_SETTING_MASK_CONTROL_OUT
UPIPE_SETTING_MASK_BULK_IN
UPIPE_SETTING_MASK_BULK_OUT
UPIPE_SETTING_MASK_INT_IN
UPIPE_SETTING_MASK_INT_OUT
UPIPE_SETTING_MASK_ISO_IN
UPIPE_SETTING_MASK_ISO_OUT

Additional names are defined that are combinations of the above:

UPIPE_SETTING_MASK_ANY
UPIPE_SETTING_MASK_ DATA_IN
UPIPE_SETTING_MASK_DATA_OUT
UPIPE_SETTING_MASK_BULKINT_IN
UPIPE_SETTING_MASK_BULKINT_OUT

These flags are described in UsbPumpInterface_EnumeratePipes().

ucPipeOrdinal If 0, the first matching pipe is used. If 1, the second matching pipe is used, and so on..

pHoldQueue A slot for holding received UBUFQEs when no physical pipe is bound. These UBUFQEs will automatically be queued to the physical pipe when a pipe is rebound to the UDATASTREAM.

pDataPlane A pointer to the parent UDATAPLANE.

usEpAddrMask Endpoint address mask, used in selecting a pipe. Bit 0 corresponds to endpoint address 0, bit 15 corresponds to endpoint address 15. Setting this mask to all ones means that any endpoint may be chosen; otherwise an endpoint may be chosen only if `(1 << endpoint_adddress) & usEpAddrMask` is non zero.

7.8 Interrupt Handling

7.8.1 UINTSTRUCT

Interrupt register structures can be used to simplify the handling of interrupts.

The function is defined as:

```
typedefPUINTSTRUCT UINTHANDLE;
typedef VOID (UINTRELOADFN) ARGS((PUINTSTRUCT));
typedef UINTRELOADFN *PUINTRELOADFN;
```

Typedef: UINTSTRUCT, *PUINTSTRUCT

```
struct TTUSB_UINTSTRUCT
   {
   UINT32  uis_magic;
   UDEVICE *uis_udev;
   PUEVENTNODE uis_reloadq;
   UEVENTNODE uis_event;
   UBUFQE  uis_uqe;
   };
```

uis_magic The signature.

uis_udev A pointer to the device.

uis_reloadq A pointer to user event queue.

uis_event Used for getting relevant events.

uis_uqe The queue element that the interrupt will use.

7.8.2 UHIL_INTERRUPT_SYSTEM_INTERFACE

The entire abstract interrupt control system is wrapped into a structure. If there are multiple logical interrupt control systems in a single computer, there will be multiple instances of this structure. Each DCD port is responsible for using these in a *reasonable* way. No slot is provided for the "interrupt system context"; instead it is assumed this will be embedded, if necessary, in a larger structure, and the interrupt subsystem will find the larger structure implicitly from the address of this structure.

The functions are defined as:

```
typedef UHIL_INTERRUPT_CONNECTION_HANDLE (UHIL_OPEN_INTERRUPT_CONNECTION_FN)
(UHIL_INTERRUPT_SYSTEM_INTERFACE *, UHIL_INTERRUPT_RESOURCE_HANDLE);

typedef BOOL    (UHIL_CLOSE_INTERRUPT_CONNECTION_FN)
   (UHIL_INTERRUPT_CONNECTION_HANDLE);

typedef BOOL    (UHIL_CONNECT_TO_INTERRUPT_FN)
   (UHIL_INTERRUPT_CONNECTION_HANDLE, UHIL_INTERRUPT_SERVICE_ROUTINE_FN *,
   VOID*);

typedef BOOL    (UHIL_DISCONNECT_FROM_INTERRUPT_FN)
   (UHIL_INTERRUPT_CONNECTION_HANDLE);

typedef BOOL    (UHIL_INTERRUPT_CONTROL_FN) (UHIL_INTERRUPT_CONNECTION_HANDLE,
   BOOL);

typedef BOOL    (UHIL_CONNECT_TO_INTERRUPT_V2_FN)
   (UHIL_INTERRUPT_SYSTEM_INTERFACE *, UHIL_INTERRUPT_CONNECTION_HANDLE,
   VOID *, UHIL_LINE_INTERRUPT_SERVICE_ROUTINE_FN *,
   UHIL_MESSAGE_INTERRUPT_SERVICE_ROUTINE_FN *,
   UHIL_MP_LINE_INTERRUPT_SERVICE_ROUTINE_FN *,
   UHIL_MP_MESSAGE_INTERRUPT_SERVICE_ROUTINE_FN *);

typedef USBPUMP_IOCTL_RESULT    (UHIL_INTERRUPT_SYSTEM_IOCTL_FN)
   (UHIL_INTERRUPT_SYSTEM_INTERFACE *, USBPUMP_IOCTL_CODE, CONST VOID *,
   VOID *);
```

The interrupt system interface structure is defined as:

```
struct UHIL_INTERRUPT_SYSTEM_INTERFACE
   {
   UHIL_OPEN_INTERRUPT_CONNECTION       *pOpenInterruptConnection;
   UHIL_CLOSE_INTERRUPT_CONNECTION      *pCloseInterruptConnection;
   UHIL_CONNECT_TO_INTERRUPT_FN         *pConnectToInterrupt;
   UHIL_DISCONNECT_FROM_INTERRUPT_FN    *pDisconnectFromInterrupt;
   UHIL_INTERRUPT_CONTROL_FN            *pInterruptControl;
   UHIL_CONNECT_TO_INTERRUPT_V2_FN      *pConnectToInterruptV2;
   UHIL_INTERRUPT_SYSTEM_IOCTL_FN       *pIoctl;
   };
```

pOpenInterruptConnection — a HIL function to open an interrupt connection handle, so we can start managing a particular IRQ. The IRQ is placed in a quieted state.

pCloseInterruptConnection — a HIL function to close an interrupt connection handle.

pConnectToInterrupt — a HIL function to connect an ISR to an interrupt. If the interrupt system permits, the interrupt is initially disabled at the interrupt controller.

pDisconnectFromInterrupt — a HIL function to disconnect the interrupt connection.

Platforms which support plug and play or device shutdown (e.g, the DOS programs which want to unhook and exit) might also have to use the disconnect-from-interrupt primitive, which disconnects an ISR without releasing the interrupt handle. This can be stubbed on embedded platforms which have no use for this.

pInterruptControl — a HIL function for manipulating the interrupt subsystem.

pConnectToInterruptV2 — a HIL function to connect an ISR to an interrupt. At most, one of the provided ISRs will be used on any given system. The caller need not provide all four routines to this service; but if the system needs to use a routine that is not provided, the connection will not be made.

In most cases, the platform decides in advance how an interrupt needs to be connected. Furthermore, uniprocessor-model interrupt processing might not be available on a given platform. For maximum portability to new systems, DCDs and HCDs should provide pMpLineIsr and pMpMessage ISR.

If the interrupt system permits, the interrupt is initially disabled at the interrupt controller.

pIoctl — a HIL function allows clients to get information about the implementation and requirements from the interrupt system for a given interrupt handle. It may be called after opening an interrupt connection handle, and can return the following information:

- whether the interrupt will connect in line mode or message mode;
- if in message mode, the number of messages that can be handled by the interrupt system.
- whether the interrupt system supports uniprocessor-model ISRs
- whether the interrupt system supports multiprocessor-model ISRs

The functions are described in detail in section 14.3.1.1, but the basic idea is this:

First, a *connection* is opened to the interrupt system, which returns a handle to be used for subsequent calls. The handle lets the connection be closed; by actually hooking the interrupt, using `pConnectToInterrupt`. It is unhooked using `pDisconnectFromInterrupt`, and changes may be made to the handling using `pInterruptControl`. The encapsulation of these functions in the

interrupt system interface structure allows multiple interrupt system implementations to coexist in the form of multiple interrupt system interface structure instances.

7.8.2.1 Initializing the UHIL_INTERRUPT_SYSTEM_INTERFACE

```
# define UHIL_INTERRUPT_SYSTEM_INTERFACE_V2 (
       /* a_pOpenInterruptConnection *  */,
       /* a_pCloseInterruptConnection * */,
       /* a_pConnectToInterrupt *       */,
       /* a_pDisconnectFromInterrupt *  */,
       /* a_pInterruptControl *         */
       /* a_pConnectToInterruptV2 *     */,
       /* a_pIoctl *                    */
       )
```

7.9 HIL Structures

7.9.1 CALLBACKCOMPLETION

Typedef: CALLBACKCOMPLETION, *PCALLBACKCOMPLETION.

Embedding Macro: CALLBACKCOMPLETIONHDR

```
struct TTUSB_CALLBACKCOMPLETION
   {
   PCALLBACKFN callback_pfn;
   VOID        *callback_ctx;
   };
```

callback_pfn the callback function, which is of the type:

```
typedef VOID CALLBACKFN (VOID * context);
```

callback_ctx the context for the function.

7.9.2 UEVENTCONTEXT

Typedef: UEVENTCONTEXT, *PUEVENTCONTEXT.

```
struct TTUSB_EVENTCONTEXT
   {
   CALLBACKCOMPLETION   **uevent_base;
   CALLBACKCOMPLETION   **uevent_head;
   CALLBACKCOMPLETION   **uevent_tail;
   CALLBACKCOMPLETION   **uevent_top;
   ULONG                uevent_overrun;
   ULONG                uevent_active;
```

```
        UPLATFORM           *uevent_pPlatform;
        };
```

uevent_base event base pointer

uevent_head event head pointer

uevent_tail event tail pointer

uevent_top event top pointer

uevent_overrun event overrun indicator

uevent_active nonzero if processing an event.

uevent_pPlatform event platform pointer

7.9.3 UPOLLCONTEXT

Typedef: UPOLLCONTEXT, *PUPOLLCONTEXT.

```
      struct TTUSB_POLLCONTEXT
        {
        PFIRMWAREPOLLFN upc_pfn;
        VOID            *upc_ctx;
        };
```

upc_pfn the polling function, which is of the type:

```
typedef VOID FIRMWAREPOLLFN (VOID *context);
```

upc_ctx the context for the function.

8. MCCI DataPump Object System

8.1 Overview of DataPump Objects

The MCCI DataPump provides a generalized set of facilities for managing and operating upon objects. Objects are data structures which have been augmented in the following ways:

- All objects are held in a central directory.
- All objects have a behavioral interface (using IOCTLs).
- All objects are arranged in a structural hierarchy that models the semantic structure of the network being implemented.

8.2 Properties of Objects

DataPump objects are used to collect and represent all the major data structures within the DataPump. This section describes the following common properties:

8.2.1 Objects Have Names

Typically the names look like normal DNS names (e.g., "msc.fn.mcci.com"). Names MCCI creates always end in ".mcci.com" but customers can do what they like. Multiple objects might have identical names, but can be distinguished by their instance numbers.

8.2.2 Objects Can Be Found By a Pointer

MCCI has library routines that can enumerate all objects using a pattern for the name with limited wild cards. For example, the user can browse for all objects matching "*.fn.*". This makes it easy for a client to match all objects of a given "kind"provided that the names follow predictable patterns.

8.2.3 Objects Have Behavior

In MCCI's object system, all objects can receive "IOCTL" operations. IOCTLs always have a common stereotype:

```
IoctlFn(pObject, IoctlCode, pInArg, pOutArg)
```

An object may choose to claim an IOCTL or not to claim it. If an object doesn't claim the IOCTL, the DataPump will try to send the IOCTL to the next logical recipient. IOCTL codes directly represent the size of the in and out arguments (as part of the numerical code).

8.2.4 Objects Have Relationships to Each Other

When an object is created, the creator specifies who the "next logical recipient" for IOCTLs should be. This next recipient is called the "IOCTL parent". IOCTLs are routable by the IOCTL system in the core DataPump, and the user can get inherited behavior. If an object doesn't supply a behavior, its IOCTL parent will be asked to provide a behavior, so the child object can inherit from its parent.

MCCI uses this to modularize the code and allow very high levels to tunnel through to very low levels.

8.3 USBPUMP_OBJECT_HEADER

USBPUMP_OBJECT_HEADER contains the basic information and maintains the information about any USB DataPump object.

USBPUMP_OBJECT_HEADER contains the following:

```
ULONG           Tag;
SIZE_T          Size;
CONST TEXT      *pName;
ULONG           InstanceNumber;
USBPUMP_OBJECT_HANDLE   Handle;
USBPUMP_OBJECT_LIST Directory;
USBPUMP_OBJECT_HEADER   *pClassParent;
USBPUMP_OBJECT_LIST ClassSiblings;
USBPUMP_OBJECT_IOCTL_FN *pIoctl;
USBPUMP_OBJECT_HEADER   *pIoctlParent;
USBPUMP_OBJECT_LIST IoctlSiblings;
UINT32          ulDebugFlags;
```

Many USB DataPump data structures are conveniently modeled with a common set of behaviors and attributes: names (which are really tuples that allow structured matching of like functions), common behaviors (modeled by IOCTLs), and static relationships (modeled as a tree).

The USBPUMP_OBJECT_HEADER structure collects the basic features into a single data structure, which can be placed at the beginning of any structure which is to be treated as an object. These behaviors and attributes can be extended by additional data that is in the structure.

Tag A unique and arbitrary tag, assigned by the object designer. It often is a number that dumps as a 4-byte ASCII constant.

Size The size of the overall structure, provided for convenience in debugging.

pName is the pointer to the object name. Object names are actually compound -- objects can have multiple names, representing the object in terms of its structure or in terms of its behavior.

InstanceNumber

Provides a unique and simple way to differentiate among multiple instances of objects with the same name.

Handle Uniquely identifies this object instance from the same instance after a DataPump restart.

Directory A list node; the object directory uses these fields.

pClassParent A pointer to the parent object.

ClassSiblings A list node that is used to chain together all the children of a given Static Parent.

pIoctl A pointer to the optional IOCTL dispatch function for this object.

pIoctlParent A pointer to the next object that is to receive any IOCTLs not claimed by this object.

ulDebugFlags

The debug flags.

8.4 USBPUMP_OBJECT_IOCTL_FN

Function: C function type for USBPUMP_OBJECT_HEADER IOCTL method functions.

Definition:

```
typedef USBPUMP_IOCTCL_RESULT
    USBPUMP_OBJECT_IOCTL_FN(
    USBPUMP_OBJECT_HEADER *pObject,
    USBPUMP_IOCTL_CODE Ioctl,
    CONST VOID *pInParam,
    VOID *pOutParam
    );
typedef USBPUMP_OBJECT_IOCTL_FN *PUSBPUMP_OBJECT_IOCTL_FN;
```

Description: Each USBPUMP_OBJECT_HEADER has an IOCTL function provided by a class-specific function associated with it.. All such functions share a common prototype, USBPUMP_OBJECT_IOCTL_FN, and should be declared in header files using that type, rather than with an explicit prototype.

For example, if a concrete object implementation defines an IOCTL function named UsbWmc_Ioctl, then the *header file* should prototype the function using:

```
USBPUMP_OBJECT_IOCTL_FN UsbWmc_Ioctl;
```

Rather than:

```
USBPUMP_IOCTL_RESULT UsbWmc_Ioctl(
USBPUMP_OBJECT_HEADER *p,
USBPUMP_IOCTL_CODE,
CONST VOID *,
VOID *
);
```

If clients wants a prototype for reference in the code, they should write the prototype *twice*:

```
USBPUMP_OBJECT_IOCTL_FN UsbWmc_Ioctl;
/* for reference, the above is equivalent to: */
USBPUMP_IOCTL_RESULT UsbWmc_Ioctl(
    USBPUMP_OBJECT_HEADER *p,
    USBPUMP_IOCTL_CODE,
    CONST VOID *,
```

```
        VOID *
        );
```

The justification for this design approach is that it highlights the fact that the function prototype is not under the control of the function implementer, but rather highlights the fact that the function is a method implementation for some class.

USBPUMP_OBJECT_IOCTL_FN must return USBPUMP_IOCTL_RESULT_NOT_CLAIMED (if the IOCTL code was not recognized), USBPUMP_IOCTL_RESULT_SUCCESS (if the IOCTL code was recognized and the operation was successfully performed), or some error code (if the IOCTL code was recognized, but the operation could not be performed for some reason).

8.5 USBPUMP_OBJECT_LIST

Function: To provide a standard, doubly-linked list component for USBPUMP_OBJECT_HEADERs.

Definition: USBPUMP_OBJECT_LIST contains:

1. USBPUMP_OBJECT_HEADER *pNext;
2. USBPUMP_OBJECT_HEADER *pPrevious;

Description: USBPUMP_OBJECT_HEADERs are likely to be on multiple lists. For consistency, rather than having many nodes named pXXXnext and pXXXlast, structure is defined that just contains the pNext and pLast for the particular list.

8.6 Derived Objects

As with the V1 DataPump, extensive use is made of type-safe derivation of objects from base object types. However, with V2, we have adopted a new methodology. This methodology depends on the facilities of C89, and results in less typing when creating derived types.

In all cases, objects that are the root of a derivation tree are defined as union types, containing a single instance of a structure that defines the contents. For example:

```
typedef struct __OBJECT_CONTENTS
   {
   ...     /* the contents of the base object */
   } OBJECT_CONTENTS, *POBJECT_CONTENTS;

typedef union __OBJECT
   {
   OBJECT_CONTENTS  Object;     /* the name “Object” will vary based on the
                                || name of the symbolic type, as appropriate.
                                */
   } OBJECT, *POBJECT;
```

A basic rule of type derivation is this: it should be possible to write the expression that names an element of an object, without knowing the concrete type that is in use. For example, it is required that if `OBJECT1`, `OBJECT2` and `OBJECT3` are all derived from `OBJECT`, the common fields in `OBJECT1`, `OBJECT2` and `OBJECT3` defined by the base type `OBJECT` will always be named `Object.`*name*. This should be true even if `OBJECT3` is based on `OBJECT2`, `OBJECT2` is in turn based on `OBJECT1`, and `OBJECT1` is based on `OBJECT`.

Two macros are conventionally defined to assist in consistently defining derived objects. It is assumed the user will create a structure and a union. The union is the top-level object, and allows the object to be viewed either in its concrete form or in its abstract form. The structure gives the concrete form contents, and must begin with the appropriate structure to reserve room for the abstract entries. Normally, the union type is named "*Object*", and the structure type is named "*Object*`_CONTENTS`". To ensure consistency, two macros are defined, called the "union prefix" and the "structure prefix" macros. Unless there is historical reason to do otherwise, these macros are always named "*Object*`_CONTENTS__UNION`" and "*Object*`_CONTENTS__STRUCT`". By convention, the "*Object*`_CONTENTS__UNION`" macro defines the same selectors that are defined by the "Object" union, and also defines an *ObjectCast* element, which allows the sub object to be directly viewed as an instance of its parent type without using a cast.

For example, suppose there is an `OBJECT` (with selector `.Object` referring to an `OBJECT_CONTENTS`), `OBJECT_CONTENTS` (containing fields `.a` and `.b`), and the macros `OBJECT_CONTENTS__UNION` and `OBJECT_CONTENTS__STRUCT`. The derived type is then defined `DERIVED_OBJECT` as follows:

```
typedef struct __DERIVED_OBJECT_CONTENTS
   {
   OBJECT_CONTENTS__STRUCT;
   UINT    c;
   VOID    *d;
   } DERIVED_OBJECT_CONTENTS, *PDERIVED_OBJECT_CONTENTS;

typedef union __DERIVED_OBJECT
   {
   OBJECT_CONTENTS__UNION;
   DERIVED_OBJECT_CONTENTS DerivedObject;
   } DERIVED_OBJECT;
```

With these definitions, suppose `pObject` points to an `OBJECT`, and `pDerived` points to a `DERIVED_OBJECT`. Then `pObject` points to the element `Object.a` and `Object.b`. `pDerived` points to the elements `Object.a`, `Object.b`, `DerivedObject.c` and `DerivedObject.d`. Furthermore, it is legal to write:

```
pObject = &pDerivedObject.ObjectCast;
```

This is the preferred way to make type-safe conversions from derived class to base class without casting.

If there is an `OBJECT3` which is derived from `OBJECT2`, `OBJECT2` is in turn derived from `OBJECT1`, and `OBJECT1` is derived from `OBJECT`:

Then use pObject = &pObject3.ObjectCast; to convert it to a pointer to `OBJECT`.

But to get a pointer to its super class, e.g., turn it to a pointer to `OBJECT2` or `OBJECT1`, cast is still needed, e.g., `((OBJECT2 *)&(pObject)->ObjectCast`.

There is an alternative method that uses "Tag" rather than `.ObjectCast`, because casts are enormously error prone. pObject must be a USBPUMP_OBJECT_HEADER and "Tag" must be the first element; the construct will create a compile time check that pObject really is pointing to an USBPUMP_OBJECT_HEADER. Thus, it is best practice to use a field "Tag"; if someone passes in (for example) a VOID*, something bad will happen at compile time; if someone passes in a pointer to the wrong object, something bad will happen at compile time; etc. The macro expansions are not intended to be easy to read; code written using them is intended to be easy to read and reasonably safe.

For example, a macro can be defined like:

```
#define  USBPUMP_OBJECT_HEADER_TO_THINGY(pObject)
((USBPUMP_THINGY *) &(pObject)->Tag)
```

8.7 MCCI Objects Hierarchy

Device Stack:

- Root object is the *default* IOCTL parent of every other object.
- Every UPLATFORM is an object; it is an IOCTL child of the root object.
- Every UDEVICE is an object; an IOCTL descendent of its UPLATFORM.
- Each protocol instance is an object, and an IOCTL child of its UDEVICE.

A client of a protocol can send a platform IOCTL to its protocol instance, and by inheritance, that IOCTL will flow down to the UPLATFORM where it gets implemented. Since UPLATFORM behavior is determined on a platform-by-platform basis, this is one of the primary ways to customize the run-time behavior of the DataPump for a specific OEM requirement. If a behavior isn't implemented, then an appropriate error code is returned to the issuer of the IOCTL.

Here is an example of how this is used. Clients want to change the behavior of the DataPump based on MMI settings (mass storage only or modem only). The platform must provide an implementation of USBPUMP_IOCTL_GET_VIDPIDMODE in the UPLATFORM-outcalls for the platform. Normally, there is a place in the OS-specific init code (e.g. the args to unucleus_UsbPumpInit ()) where the client can pass a pointer to an IOCTL function. As soon as this behavior is implemented, DataPump will automatically start tracking the MMI setting.

8.8 MCCI Objects Functions

8.8.1 UsbPumpObject_Ioctl

Function: Dispatch an IOCTL through a DataPump Object, given the object header.

Definition:

```
USBPUMP_IOCTL_RESULT UsbPumpObject_Ioctl(
USBPUMP_OBJECT_HEADER    *pHdr,
USBPUMP_IOCTL_CODE   Request,
CONST VOID      *pInBuffer,
VOID        *pOutBuffer,
);
```

Description: This routine issues an IOCTL in the standard way via the object header given at *pHdr. If pHdr is NULL, or if pHdr->pIoctl is NULL, then the request is failed with NOT_CLAIMED status. Otherwise the request is passed in, and the client gets to handle it. In order to avoid excessive stack depth, this routine walks the object tree upwards until a result is obtained. This way, there's no need for object methods to insert extra code to pass unclaimed IOCTLs down.

On the other hand, it means that an inline version of this function is not provided.

Returns: USBPUMP_IOCTL_RESULT_NOT_CLAIMED if nobody claimed the IOCTL.

USBPUMP_IOCTL_SUCCESS for success and some other error code if failure.

8.8.2 UsbPumpObject_Init

Function: Fills in an object header, and registers the object with the appropriate authorities.

Definition:

```
VOID UsbPumpObject_Init(
USBPUMP_OBJECT_HEADER *  pNew,
USBPUMP_OBJECT_HEADER *  pClassHeader,
UINT32              Tag,
SIZE_T              Size,
CONST TEXT *        pName,
USBPUMP_OBJECT_HEADER *  pIoctlParent,
USBPUMP_OBJECT_IOCTL_FN *    pIoctlFn
);
```

Description: The object header at pNew is initialized with the information passed in from the arguments.

Returns: No explicit result.

8.8.3 UsbPumpObject_DeInit

Function: Un-registers an object.

```
VOID UsbPumpObject_DeInit(
USBPUMP_OBJECT_HEADER *pObject
);
```

Description: The object is de-registered. A multiple de-init is checked for: The link fields are cleared after deregistering. A dual de-register causes a software-check to be issued. It is important to leave enough linkage in place to allow the UPLATFORM to be found, in case it is necessary to display a message.

Returns: No explicit result.

8.8.4 UsbPumpObject_EnumerateMatchingNames

Function: Scans a directory (given by the specific object) looking for the next matching name.

Definition:

```
USBPUMP_OBJECT_HEADER *UsbPumpObject_EnumerateMatchingNames(
USBPUMP_OBJECT_HEADER *pClassObject,
CONST TEXT *pPattern,
USBPUMP_OBJECT_HEADER *pLastObject OPTIONAL
);
```

Description: This function is a wrapper for the class directory mechanisms. It allows a simple traversal of the matching objects, using a loop of the form:

```
p = NULL;
while ((p = UsbPumpObject_EnumerateMatchingNames(pDirObj, pPat, p))
!= NULL)
{
// process p
}.
```

Returns: Pointer to next object in sequence, or NULL.

8.8.5 UsbPumpObject_FunctionOpen

Function: OS-specific driver will call this function to make a connection to leaf object.

Definition:

```
USBPUMP_OBJECT_HEADER *UsbPumpObject_FunctionOpen(
USBPUMP_OBJECT_HEADER *pFunctionObject,
USBPUMP_OBJECT_HEADER *pClientObject,
VOID *pClientContext,
USBPUMP_IOCTL_RESULT *pIoctlResult OPTIONAL
);
```

Description: USBPUMP_IOCTL_FUNCTION_OPEN is sent from an OS-specific driver to a leaf object to notify the leaf object that a client is about to begin I/O.

The OS-specific driver must prepare an OS-specific driver

USBPUMP_OBJECT_HEADER, which is then registered with the leaf object. It returns the actual object handle (normally the same object as was opened), which is to be used for I/O.

Return: Pointer to opened object, or NULL. If pIoctlResult is not NULL, *pIoctlResult is set to the IOCTL result code.

8.8.6 UsbPumpObject_FunctionClose

Function: Closes a previously opened object pointer.

Definition:

```
BOOL UsbPumpObject_FunctionClose(
USBPUMP_OBJECT_HEADER *pIoObject,
USBPUMP_OBJECT_HEADER *pClientObject,
USBPUMP_IOCTL_RESULT *pResult OPTIONAL
);
```

Description: This call reverses a previous open. pIoObject must have been returned by a previous call to ..._FunctionOpen; pClientObject must match what was passed at open time.

Return: TRUE for success, FALSE for failure.

8.8.7 UsbPumpObject_GetDevice

Function: Given some object, it finds the underlying UDEVICE.

Definition:

```
UDEVICE *UsbPumpObject_GetDevice(
USBPUMP_OBJECT_HEADER *pObject
);
```

Description: This routine is a wrapper for USBPUMP_IOCTL_GET_UDEVICE.

Return: Pointer to underlying UDEVICE, or NULL if none such exists under this object.

8.8.8 UsbPumpObject_GetRoot

Function: Given some object, this function is called to find the root object of MCCI object registry.

Definition:

```
USBPUMP_OBJECT_ROOT *UsbPumpObject_GetRoot(
USBPUMP_OBJECT_HEADER *pObject
);
```

Description: This routine is a wrapper for USBPUMP_IOCTL_GET_ROOT.

Return: Pointer to root object, or NULL.

8.8.9 UsbPumpObject_RootInit

Function: Initializes a freshly created root object.

Definition:

```
BOOL UsbPumpObject_RootInit(
USBPUMP_OBJECT_ROOT *pRoot,
UPLATFORM *pPlatform
);
```

Description: This routine is called early during initialization. It initializes the root object's header entries (reflexively), then makes sure the root object has a pointer to a UPLATFORM for use, e.g., in doing memory allocations.

This function also sets up an IOCTL method for the root object.

Return: TRUE for success, FALSE for failure.

8.8.10 UsbPumpObject_FindAndSetDebugFlags

Function: Finds all size information from the root table.

Definition:

```
BYTES UsbPumpObject_FindAndSetDebugFlags (
        USBPUMP_OBJECT_HEADER *     pDirObject,
        CONST TEXT *        pPattern,
```

```
    UINT32          ulFlagsMask,
    UINT32          ulFlagsBits
    );
```

Description: Scans a directory (given by the specific object) looking for the next matching name and sets debug flags for a matched object.

Returns: No explicit result.

8.8.11 UsbPumpObject_SetDebugFlags

Function: Sets debug flags for a given object.

Definition:

```
BYTES UsbPumpObject_SetDebugFlags (
    USBPUMP_OBJECT_HEADER *     pObjectHeader,
    UINT32 ulDebugFlags
    );
```

Description: This routine sets debug flags for the specified pObjectHeader.

Returns: No explicit result.

8.8.12 UsbPumpObject_GetDebugFlags

Function: Gets debug flags for a given object.

Definition:

```
BYTES UsbPumpObject_GetDebugFlags (
    USBPUMP_OBJECT_HEADER       *pObjectHeader                  );
```

Description: This routine returns debug flags for given pObjectHeader.

Returns: Debug flags.

9. MCCI IOCTL Handling

IOCTLs form the basic method of abstract inter-module messaging in the V1.x DataPump. While extremely convenient, they suffer from one drawback in the V2.0 environment: they are synchronous. Numerous changes in the hardware environment make it preferable to have an asynchronous variant of the same API. At the same time, it is undesirable to break older code or force the addition of new API entry points for every object. As is usual in this kind of situation, the API is extended in a few orthogonal directions, while allowing the original API providers and

clients to operate unchanged. For simplicity, asynchronous IOCTLs are only allowed to be issued to objects.

9.1 Codes for Asynchronous IOCTLs

The naming convention for appending "…_ASYNC" to asynchronous IOCTL codes and related macros is adopted. For example, USBPUMP_IOCTL_CODE USBPUMP_IOC_ASYNC () is defined as a variant of USBPUMP_IOCTL_CODE USBPUMP_IOC (), that creates an IOCTL code of the same general kind, but one that is asynchronous.

For convenience, the following lists all IOCTL codes:

```
USBPUMP_IOCTL_CODE USBPUMP_IOC_ASYNC(
        UINT32 DirMask,
        UINT8  RawGroup,
        UINT8  Num,
        UINT   ArgSize
        )

USBPUMP_IOCTL_CODE USBPUMP_IOCTL_VOID_ASYNC(
`       UINT8 Group,
        UINT8 Num
        );

USBPUMP_IOCTL_CODE USBPUMP_IOCTL_R_ASYNC(
        UINT8 Group,
        UINT8 Num,
        ArgumentTypeExpression
        );

USBPUMP_IOCTL_CODE USBPUMP_IOCTL_W_ASYNC(
        UINT8 Group,
        UINT8 Num,
        ArgumentTypeExpression
        );

USBPUMP_IOCTL_CODE USBPUMP_IOCTL_RW_ASYNC(
        UINT8 Group,
        UINT8 Num,
        ArgumentTypeExpression
        );
```

The argument type expression is for the body of the IOCTL, as before. However, the actual IOCTL body that is allocated for an asynch IOCTL must always have a header. Therefore, a VOID_ASYNC IOCTL still requires a buffer when it is being issued; this buffer contains the header and routing information. This buffer is in addition to the IOCTL argument itself.

9.2 The Meaning of "Synchronous IOCTL"

A synchronous IOCTL is one that can never block. It therefore may be called via either the synchronous IOCTL API or the asynchronous IOCTL API. If a client issues a synchronous IOCTL via the synchronous API, all will go well. If a client issues a synchronous IOCTL via the asynchronous API, all will go well, but (in addition) the outer asynchronous API will arrange to call the callback function when the synchronous API completes.

On the other hand, if a client issues an asynchronous IOCTL via a synchronous API, the synchronous API will immediately reject the IOCTL, with an appropriate error code.

9.3 The Meaning of an Asynchronous IOCTL

An asynchronous IOCTL is one that might have to block – in other words, it might have to schedule execution later. Therefore, the caller of an asynch IOCTL must do three extra things:

1. Allocate a block of memory (not on the stack) which will be used for tracking the progress of the IOCTL

2. Allocate the input and output memory buffers for the parameters, if any – again, not from the stack.

3. Code a function that will be called back when the IOCTL finishes. This function will be passed a user-specified context pointer, along with a pointer to the tracking object, and the result returned by the IOCTL.

After doing this, the IOCTL can be issued much as before; but there will be no immediate status returned.

9.4 USBPUMP_IOCTL_QE – The Asynchronous IOCTL Tracking Block

This QE is used to track the progress of the IOCTL through the system, and to arrange for the client's completion to be called. The completion routine is always called in DataPump context, but it might be called in one of two ways:

1. Directly from the point of completion, if possible ("fast callback") or,

2. Indirectly, from the DataPump event loop ("safe callback")

The first use pattern is useful in situations where limited stack growth is expected; it is faster. The second use pattern is used in situations where the stack might grow to an unbounded degree; it is safer. However, the second use pattern directly requires that the DataPump event loop be accessible, and therefore restricts use of asynchronous IOCTLs to within the DataPump task.

For simplicity, only one form of the QE is used, holding enough storage to allow either style of callback to be used. This allows the DataPump core to force the "safe" policy dynamically if it needs to. Clients cannot tell which way has been used, apart from stack depth.

Unfortunately, the layout of the QE has to be in scope to allow clients to pre-allocate them. However, the APIs are designed to allow a certain amount of isolation on the client's side, similar to how things work with the IOCTL ARG macros.

9.5 Asynchronous Handling Logic in the DataPump

When an asynch IOCTL is processed, the downward path operates more or less in the same way that the synchronous path works. The differences are:

- If it is really a synchronous call, the synchronous path is followed directly (without using the `USBPUMP_IOCTL_QE`) and then the completion routine is called (according to the safety policy).

- A special internal (synchronous) IOCTL is used to pass the `USBPUMP_IOCTL_QE` into the synchronous IOCTL entry point. As the IOCTL moves down the chain, `pIoctlQe->pCurrent` is modified to point to the current object. As before, the IOCTL handlers have the choice of ignoring the special IOCTL code (not claiming it), rejecting it (by returning an error code), tunneling into the `USBPUMP_IOCTL_QE` to get the code and again do the same handling (not claiming it, rejecting it, or claiming it).

- If an asynch IOCTL is claimed without error, the driver assumes that the processing is done for now, and that the IOCTL handler will do the work of completing. If it is claimed with error, then the driver completes the IOCTL. Otherwise (if not claimed), the process repeats with the next device in the IOCTL chain.

- Later, when the object's IOCTL handler gets called back from whatever reason it blocked, it can simply complete the `USBPUMP_IOCTL_QE`. At present there is no way to send the queue element itself further down the stack. However, nothing prevents the object's IOCTL handler from itself sending an asynch IOCTL further down the stack.

- Object IOCTL handlers explicitly complete `USBPUMP_IOCTL_QE`s by calling `UsbPumpIoctlQe_Complete(pPlatform, pIoctlQe, result)`. From that moment, the handler must no longer refer to the QE; it may already have been reused.

- `UsbPumpIoctlQe_Complete()` examines the result; if the result is NOT_CLAIMED, then either `UHIL_swc()` is called, or passing the request towards the root (the implementation decision has not yet been made). Otherwise, if `safeCompletionPolicy(pIoctlQe)` is set - i.e. if `pIoctlQe >Callback.callback_callback_ctx` is non-NULL, the result is saved, and `UsbPumpIoctlQe_Complete()` calls `UsbPumpPlatform_PostIfNotBusy`) on the `pIoctlQe`.Callback, scheduling `UsbPumpIoctlI_EventComplete()`, which in turn calls the client's `pDoneFn` (from within the DataPump thread). Otherwise, `UsbPumpIoctl_Complete()` calls the client's pDoneFn directly.

9.6 Asynchronous IOCTL API Functions

There are four functions provided to facilitate use of asynchronous IOCTLs;

1. UsbPumpObject_IoctlAsync (),
2. UsbPumpIoctlQe_Cancel(),
3. UsbPumpIoctlQe_SetCancelRoutine(), and
4. UsbPumpIoctlQe_Complete().

9.6.1 UsbPumpObject_IoctlAsync

Use this function to submit an asynchronous IOCTL to a DataPump object.

```
VOID
UsbPumpObject_IoctlAsync(
          USBPUMP_OBJECT_HEADER    *pHdr,
          USBPUMP_IOCTL_CODE        Request,
          CONST VOID               *pInBuffer,
          VOID                     *pOutBuffer,
          USBPUMP_IOCTL_QE         *pRequest,
          RECSIZE                   SizeRq,
          USBPUMP_IOCTL_QE_DONE_FN *pDoneFn,
          VOID                     *pDoneCtx,
          BOOL                      fSafe
          );
```

The first four parameters of this function are exactly as for `UsbPumpObject_Ioctl()`. Subsequent parameters are used for controlling the asynchronous processing, as follows.

- `pRequest` is a pointer to a `USBPUMP_IOCTL_QE`, which is used to track the progress of the request.
- `SizeRq` should be `sizeof(*pRequest)`, and is used for version-matching purposes.
- `pDoneFn` is a pointer to the user's completion function (see below).
- `pDoneCtx` is an arbitrary client-supplied context pointer.
- `fSafe`, if `TRUE`, forces the completion to be deferred by posting an event to the event queue. If `FALSE`, the DataPump *may* elect to call `pDoneFn` directly at the point where the IOCTL is completed. However, there is no guarantee that the DataPump will not elect to defer the completion even if `fSafe` is `FALSE`.

The completion function should have the signature

```
typedef VOID
USBPUMP_IOCTL_QE_DONE_FN(
```

```
        USBPUMP_IOCTL_QE *pIoctlQe,
        VOID *pDoneInfo,
        USBPUMP_IOCTL_RESULT result,
        VOID *pOutArg
        );
```

9.6.2 UsbPumpIoctlQe_Cancel

This routine attempts to cancel a pending asynchronous IOCTL.

```
VOID
UsbPumpIoctlQe_Cancel(
        USBPUMP_IOCTL_QE *pRequest
        );
```

This routine (which must be called from DataPump thread context) sets the "cancelled" flag in the `USBPUMP_IOCTL_QE`, and calls the current cancel routine (if any) for that `USBPUMP_IOCTL_QE`. There is no guarantee that this will have any net effect, but properly coded processing routines will set a cancel function whenever the `USBPUMP_IOCTL_QE` is sitting in a queue, so that it can be removed and completed.

If the current cancel function is `NULL`, this function simply sets the cancel flag in `pRequest`. Presumably in this case, the layer that currently is processing `pRequest` will arrange to check the cancel flag and abort processing if necessary. (Alternately, the request might already be waiting to be completed via the event queue.)

9.6.3 UsbPumpIoctlQe_SetCancelRoutine

This routine sets or clears the cancel routine pointer in an `USBPUMP_IOCTL_QE`.

```
BOOL
UsbPumpIoctlQe_SetCancelRoutine(
        USBPUMP_IOCTL_QE *pRequest,
        USBPUMP_IOCTL_QE_CANCEL_FN *pCancelFn,
        VOID *pCancelContext
        );
```

This routine sets the cancel function and cancel information for this queue element. It returns `TRUE` if the cancel function was non-`NULL`, and is now `NULL`.

This function should never be called by the layer that issues the request. It is to be called by the layer that is processing the request. If the request is being placed in a queue, the processing layer should use `UsbPumpIoctlQe_SetCancelRoutine()` to set up a cancellation routine that will remove the element from the queue and complete it immediately. Later, when the processing layer removes the request from the queue, the processing layer should use `UsbPumpIoctlQe_SetCancelRoutine()` to set the cancellation routine to `NULL`.

9.6.4 UsbPumpIoctlQe_Complete

This routine marks an `USBPUMP_IOCTL_QE` for completion, potentially calling the completion function directly.

```
VOID
UsbPumpIoctlQe_Complete(
          UPLATFORM *pPlatform /* OPTIONAL */,
          USBPUMP_IOCTL_QE *pRequest,
          USBPUMP_IOCTL_RESULT Result
          );
```

Though this may be thought of as a method function of the IOCTL, the "more global" pointer is put first in the parameter list.

`pPlatform` is optional because `UsbPumpIoctlQe_Complete` will simply locate the platform pointer if `pPlatform` is `NULL`. However, this is an expensive operation, so high-frequency paths should supply a pre-computed platform pointer.

9.7 Asychronous IOCTL Processing Patterns

Code that processes asynchronous IOCTLs must add a few extra checks. Following is some sample code for an IOCTL processing function. The sample function has to process two IOCTLs, one is synchronous and the other asynchronous:

```
USBPUMP_IOCTL_RESULT
Sample_Ioctl(
          USBPUMP_OBJECT_HEADER *pObject,
          USBPUMP_IOCTL_CODE Ioctl,
          CONST VOID *pInParam,
          VOID *pOutParam
          )
          {
          USBPUMP_IOCTL_RESULT Result;
          USBPUMP_IOCTL_QE *   pIoctlQe;

          /*
          || the following code is only needed if this routine is to process
          || async IOCTLs.  It validates the wrapper and updates the pInParam
          || and pOutParam pointers to they point to the actual parameters
          || passed in the wrapper.
          */
          if (USBPUMP_IOCTL_IS_ASYNC(Ioctl))
              {
              pIoctlQe = (USBPUMP_IOCTL_QE *) pInParam;
              if (pIoctlQe == NULL)
                  {
                  return USBPUMP_IOCTL_RESULT_IN_PARAM_NULL;
```

```
                }

            pInParam = pIoctlQe->pInArg;
            pOutParam = pIoctlQe->pOutArg;
            }
        else
            {
            /*
            || synchronous: pIoctQe is not used, but we must keep compilers
            || happy by giving it a value.
            */
            pIoctlQe = NULL;
            }

        /* assume that the IOCTL will not be recoqnized -- simplifies code below */
        Result = USBPUMP_IOCTL_RESULT_NOT_CLAIMED;

        /* dispatch the IOCTL */
        switch (Ioctl)
            {
        case USBPUMP_IOCTL_SAMPLE_SYNCHRONOUS_REQUEST:
            {
            /* often, only one of the following is used, but this is sample code */
            CONST USBPUMP_IOCTL_SAMPLE_SYNCRHONOUS_REQUEST_ARG * CONST
                pInArg = pInParam;
            USBPUMP_IOCTL_SAMPLE_SYNCHRONOUS_REQUEST_ARG * CONST
                pOutArg = pOutParam;

            /*
            || Mandatory:  check the input and output pointers before
            || attempting to use them.
            */
            Result = UsbPumpIoctl_CheckParams(Ioctl, pInParam, pOutParam);

            /*
            || If successful, proceed with the IOCTL processing; otherwise
            || fail the request.
            */
            if (USBPUMP_IOCTL_SUCCESS(Result))
                {
                /* ... do the sample processing and set Result */
                }
            }
            break;

        case USBPUMP_IOCTL_SAMPLE_REQUEST_ASYNC:
            {
/*          CONST USBPUMP_IOCTL_SAMPLE_REQUEST_ASYNC_ARG * CONST
||              pInArg = pInParam;
```

```
*/
            USBPUMP_IOCTL_IOCTL_SAMPLE_REQUEST_ASYNC_ARG * CONST
                pOutArg = pOutParam;

            /*
            || Mandatory:  check the input and output pointers before
            || attempting to use them.
            */
            Result = UsbPumpIoctl_CheckParams(Ioctl, pInParam, pOutParam);

            /*
            || If successful, proceed with the IOCTL processing; otherwise
            || fail the request.
            */
            if (USBPUMP_IOCTL_SUCCESS(Result))
                {
                if (/* can complete request now */)
                    {
                    UsbPumpIoctlQe_Complete(
                        pPlatform,
                        pIoctlQe,
                        Result
                        );
                else
                    {
                    /* do something that will cause a callback later */
                    ...
                    /* tell IOCTL dispatcher that we accepted the IOCTL */
                    Result = USBPUMP_IOCTL_RESULT_SUCCESS;
                    }
                }
            else
                {
                /* failed: complete the request */
                UsbPumpIoctlQe_Complete(
                    pOtgFsm->pPlatform,
                    pIoctlQe,
                    Result
                    );

                /* Result is not USBPUMP_IOCTL_RESULT_NOT_CLAIMED, so dispatcher
                || will decide that the IOCTL was claimed.
                */
            }
            break;

        ... other cases

        /* unrecognized ioctls */
```

```
        default:
            /* Result is already USBPUMP_IOCTL_RESULT_NOT_CLAIMED, do nothing */
            break;
            }   /* end switch */

        return Result;

    }
```

10. MCCI Event Handling

10.1 Event Support Function

10.1.1 UsbAddEventNode

Function: Add an event node to an event node queue.

Definition:

```
#include "ueventnode.h"

    VOID UsbAddEventNode(
        UEVENTNODE **ppHead,
        UEVENTNODE *pNode,
        UEVENTFN *pfn,
        VOID *info
        );
```

Description: The specified node is appended to the specified queue of event nodes, and the context field of the node is set to `info`.

Returns: Nothing.

10.1.2 UsbReportEvent

Function: Runs an event queue, and reports events to all who register before

Definition:

```
#include "usbpumpapi.h"

    VOID UsbReportEvent (
        UDEVICE *dev,
        UEVENTNODE *head,
        UEVENT why,
        VOID *info
        );
```

Description: The list is run and the event is reported to each member of the list; centralized for maintenance purposes.

There are a few subtle occurrences that take place which allow the event processing routine to delete itself from the list, even if it is at the head of the list; however, if a new event processing routine is added to the list while the list is being run it is not specified whether or not it will be visited. It depends on whether the new node is added into the list effectively before, at, or after the node being visited.. If it is inserted before or at the node to be visited, the visit will not occur, because the node's forward link is observed before a visit actually takes place. (This allows deletions to work.) The tail of the list is observed at startup time; therefore a node must not be deleted unless it is being visited ; otherwise the event might never terminate.

Returns: Nothing.

Notes: This routine should only be used for user-defined event queues. Events should be posted to queues internal to the DataPump using higher-level routines.

10.2 Event Handling In Pre-2.0 DataPump

10.2.1 UHIL_DoEvents

Function: Drive the event dispatcher.

Definition:

```
VOID UHIL_DoEvents(
    PUEVENTCONTEXT pevq,
    ULONG max_events
    );
```

Description: This routine removes events from the event queue and passes them to the dispatcher routine. The parameter `max_events` controls the 'loop' behavior of this routine. If set to 0, this routine will continue looping as long as events remain to be processed. Otherwise, it will return after processing at most `max_events` events. It will always return when no more events remain to be processed.

Returns: Nothing.

Notes: This function may dispatch events into the USB DataPump. Its implementation should always call UHIL_dispatchevent() to do the work, because it will insert any required wrapper logic to protect against interrupt routines.

10.2.2 UHIL_DoPoll

Function: UHIL Firmware Polling routine.

Definition:

```
BOOL UHIL_DoPoll(
    PUPOLLCONTEXT ppoll,
    PUEVENTCONTEXT peq
    );
```

Description: If a firmware polling routine is registered, it is called. The debug output polling routine is also called.

Returns: TRUE: at least one event is queued. FALSE: no events are queued.

10.2.3 UHIL_PostEvent

Function: Post a new event to the event queue for task-time processing.

Definition:

```
VOID UHIL_PostEvent(
    PUEVENTCONTEXT pevq,             -- event queue
    CALLBACKCOMPLETION *pCompletion  -- the completion to be run
    )
```

Description: This function is a bridge between the portable code and the operating-system-specific code that provides the DataPump's event queue.

The specified CALLBACKCOMPLETION is queued for later processing in DataPump context. The CALLBACKCOMPLETION contains at least two fields, which must have been initialized by the caller:

- callback_pfn points to a function of type CALLBACKFN, with prototype:
  VOID ()(CALLBACKCOMPLETION *);
- callback_ctx is a VOID* pointer that may be used for any purpose.

The caller may also define a structure derived from CALLBACKCOMPLETION carrying extra information, using the macro CALLBACKCOMPLETIONHDR.

For example:

```
struct my_callback_completion
    {
    CALLBACKCOMPLETIONHDR;    // must be first
    int foo;
```

```
struct bar *baz;
char something[32];
};
```

A pointer to this object can then be cast safely to a PCALLBACKCOMPLETION for the purpose of passing through the event handling package.

Returns: Nothing.

Notes: This function is intended only to be used by the porting engineer and by MCCI for implementing the datapump framework. It is not to be used in other situations. Instead, use UHIL_PostCallback().

If the event queue overflows, events will be lost.

It is a programming error to have multiple instances of the same CALLBACKCOMPLETION structure in the queue multiple times.

UHIL_PostEvent may be implemented as a macro. Therefore, the arguments may be evaluated more than once. To avoid this, call UHIL_PostEvent using:

```
(UHIL_PostEvent)(pevq, pcb);
```

This will avoid any macro expansions.

UHIL_PostEvent() may be called in any context.

Returns: Nothing.

10.2.4 UHIL_ReleaseEventLock

Function: Release the event lock by setting uevent_active to 0 after running any deferred events.

Definition:

```
#include "udevhil.h"

VOID UHIL_ReleaseEventLock(
    UEVENTCONTEXT *pvt
    );
```

Description: Event lock is the field of uevent_active in EVENTCONTEXT which indicates events are being processed if it not zero. Refer to Section 7.5.1 UEVENT.

Assuming the caller holds the lock on the event lock, release the lock; but first run all events.

Returns: No explicit result.

10.2.5 UHIL_CheckEvent

Function: Determines if an event is waiting in the event queue.

Definition:

```
CALLBACKCOMPLETION *UHIL_CheckEvent(
    PUEVENTCONTEXT pevq
    );
```

Description: This function is a bridge between the portable code and the operating-system implemementation of the event queue. This API examines the queue to see if there are any events waiting to be processed; if so, the first one is returned. However, the event is NOT removed from the queue.

Notes: This function is intended only to be used by the porting engineer and by MCCI for implementing the datapump framework. It is not to be used in other situations.

UHIL_CheckEvent may be implemented as a macro. Therefore, the arguments may be evaluated more than once. To avoid this, call UHIL_CheckEvent using:

(UHIL_CheckEvent)(pevq);

This will avoid any macro expansions.

UHIL_CheckEvent() must be called in DataPump context.

Returns: A pointer to the next event or NULL if no events are queued.

10.2.6 UHIL_dispatchevent

Function: Dispatches a UHIL event into the DataPump.

Definition:

```
BOOL UHIL_dispatchevent(
    PUEVENTCONTEXT pevq,
    CALLBACKCOMPLETION *pev
    );
```

Description: The completion is unwrapped and dispatched.

Returns: TRUE if the pointer was null and the function pointer was null.

10.2.7 UHIL_GetEvent

Function: Get the next event from the event queue.

Definition:

```
CALLBACKCOMPLETION *UHIL_GetEvent(
    PUEVENTCONTEXT *pevq
    );
```

Description: This function is a bridge between the portable code and the operating-system-specific code that provides the DataPump's event queue. UHIL_GetEvent returns the next event from the event queue, removing it from the queue. If no events are available in the queue, it returns immediately.

Notes: This function is intended only to be used by the porting engineer and by MCCI for implementing the datapump framework. It is not to be used in other situations.

For a blocking API, refer to Section UHIL_GetEventEx().

This function may be implemented as an inline macro. As such, the argument pevq may be evaluated more than once. However, a functional version is available in the library; the code can choose to use it by calling

(UHIL_GetEvent)(...)

The parenthesization defeats recognition of the macro.

UHIL_GetEvent() must be called in data-pump context.

If bWait is TRUE, there may be platform-specific restrictions on the context from which UHIL_GetEventEx() may be called.

Returns: A pointer to the next event to be processed, or NULL.

10.2.8 UHIL_GetEventEx

Function: Get the next event from the event queue, possibly waiting.

Definition:

```
CALLBACKCOMPLETION *UHIL_GetEventEx(
        PUEVENTCONTEXT *pevq,
        BOOL bWait
        );
```

Description: This function is a bridge between the portable code and the operating-system-specific code that provides the DataPump's event queue. UHIL_GetEventEx

returns the next event from the event queue, removing it from the queue. If no events are available in the queue, the value of bWait is consulted. If bWait is FALSE, UHIL_GetEventEx() returns immediately; otherwise UHIL_GetEventEx() delays (in a reasonably efficient, OS-specific method) until an event is available.

Notes: This function is intended only to be used by the porting engineer and by MCCI for implementing the datapump framework. It is not to be used in other situations.

This function may be implemented as an inline macro. As such, the arguments may be evaluated more than once. However, a functional version is available in the library; the code used can choose to use it by calling:

(UHIL_GetEventEx)(...)

The parenthesization defeats recognition of the macro.

UHIL_GetEventEx() must be called in data-pump context.

If bWait is TRUE, there may be platform-specific restrictions on the context from which UHIL_GetEventEx() may be called.

Returns: A pointer to the next event to be processed, or NULL.

10.2.9 UHIL_PostEvent_GetEventStatus

Function: Posts a new event to the event queue for task-time processing.

Definition:

```
BOOL UHIL_PostEvent_GetEventStatus(
    PUEVENTCONTEXT      pEventCtx,
    PCALLBACKCOMPLETION pCallBack
    );
```

Description: It is first ascertained the interrupts are disabled, to prevent reentrancy. The event information is then saved in the event queue at the slot pointed to by the queue tail pointer.

After saving the event information, an attempt is made to advance the tail pointer to the next slot. If an advanced tail pointer will be past the end of the array that constitutes the event queue, the tail is set to the start of the array.

If the advancing tail will cause the tail to point to the same cell as head, it is not advanced; instead, an overrun counter is incremented.

Returns: TRUE if event queue was empty, FALSE otherwise.

10.2.10 UsbPostIfNotBusy

Function: Posts a callback completion, unless it has already been posted.

Definition:

```
BOOL UsbPostIfNotBusy(
    UDEVICE *pSelf,
    CALLBACKCOMPLETION *pCompletion,
    VOID *pContext
    );
```

Description: If the completion routine is cooperative, this routine can be used to compress completion events into a single dispatch of the completion function.

Notes: This implementation is extremely primitive, and requires that all calls to UsbPostIfNotBusy () use the same context pointer.

The UDEVICE is added in anticipation of a UHIL handle being added to the UDEVICE, and then required for all UHIL calls.

Because this might be used by an interrupt handler, it's necessary to guard the update.

Returns: TRUE if the routine was newly scheduled; FALSE if it was already pending.

10.2.11 UsbMarkCompletionBusy

Function: Mark CALLBACKCOMPLETION is busy.

Definition:

```
#define UsbMarkCompletionBusy(
        /* UDEVICE *             */ pDevice,
        /* CALLBACKCOMPLETION *  */ pCompletion,
        )
```

Description: This is called before call UHIL_PostCallback () to set CALLBACKCOMPLETION is busy. It can be used to compress completion events into a single dispatch of the completion function.

Returns: TRUE if old value of *pCompletion was NULL, FALSE otherwise.

10.2.12 UsbMarkCompletionNotBusy

Function: Mark CALLBACKCOMPLETION is not busy.

Definition:

```
#define UsbMarkCompletionNotBusy(
        /* UDEVICE *            */ pDevice,
        /* CALLBACKCOMPLETION *  */ pCompletion
        )
```

Description: This is called before call UHIL_PostCallback () to clear CALLBACKCOMPLETION is busy. It can be used to compress completion events into a single dispatch of the completion function.

Returns: None.

10.2.13 UHIL_SynchronizeToDataPumpDevice

Function: Try to grab the DataPump (if it is not already busy); or else queue a function to be called from the event queue.

Definition:

```
VOID UHIL_SynchronizeToDataPumpDevice(
    UDEVICE *pDevice,
    USBPUMP_SYNCHRONIZATION_BLOCK *pSynchBlock,
    USBPUMP_SYNCHRONIZED_FN *pSynchFn,
    VOID *Context1,
    VOID *Context2
    );
```

Description: It is often convenient to go directly into the DataPump code from external context, grabbing the existing thread to run DataPumpcode (avoiding a context switch). This function can be used to provide a simple way to do this, given a pointer to a UDEVICE and a small block of RAM.

If the DataPump is not in use, it is entered, calling the pSynchFn directly, and then running the event queue until it empties. Otherwise, an event is queued to the DataPump and return. When the event is run, the pSynchFn will be called.

Context1 and Context2 are used to save some information which will be used in pSynchFn, e.g., VbusState, ClockStatus.

Notes: If there are multiple nested incalls to the datapump, any inner nested calls will defer, but will be processed by the embedded event processing loop in this

function. Use of this routine requires a few changes to UHIL_ functions that would not otherwise be necessary:

UHIL_PostEvent () should check the active count, and not activate the USB thread unless there is no concurrent activity body in the datapump.

The USB thread needs a way to atomically grab the datapump or go to sleep (so UHIL_PostEvent () will always wake it up if there is something to do).

It is conceivable that the user would not elect to run datapump calls directly from here. In such circumstances, the user would choose to replace this routine with one that always queues the block.

There is no convenient way to limit the amount of time spent in the DataPump with this call; so APIs called from interrupt level probably should not use this mechanism.

On a multiprocessor system (e.g., as a WDM driver), this would need to be replaced by code using something that is more multiprocessor safe (e.g., the queued work blocks of MCCIWDM).

It is important to ensure the SYNCHRONIZATION_BLOCK is zeroed before its first use. After that, we'll maintain it. For a given synchronization block, only one pSynchFn should be used. (It is acceptable to use one pSynchFn with many synch blocks, but not to use many different pSynchFns with a single synch block, changing over time.) It could be argued that the API should provide an "initialize synch block" function that would set the function pointer at init time. However, that would require another function and more initialization -- this way, zeroing the block is enough to initialize it. Also, there is almost never more than one place where a given synchronization block is actually used -- so initializing separately would actually increase the code size.

Returns: No explicit result.

10.3 Event Handling in Post-2.0 DataPump

Event posting and handling in the pre-2.0 DataPump was very much DCD-centric. This made sense when all events were likely to be created by the device controller hardware in response to interrupts.

With the host stack, the situation has changed. Rather than create a parallel set of HCD-centric event APIs, MCCI has chosen to create a set of UPLATFORM-based APIs. These can then be used as needed by the various components of the DataPump stack.

A summary of the new APIs is given in Table 10.

Table 10. New Event-Handling APIs

API Name	Description
UsbPumpPlatform_CheckEvent	A variant of UHIL_ that takes a UPLATFORM pointer.
UsbPumpPlatform_DoEvents	A variant of UHIL_DoEvents that takes a UPLATFORM pointer.
UsbPumpPlatform_DispatchEvent	A variant of UHIL_dispatchevent that takes a UPLATFORM pointer.
UsbPumpPlatform_GetEvent	A variant of UHIL_ that takes a UPLATFORM pointer
UsbPumpPlatform_GetEventEx	A variant of UHIL_GetEventEx that takes a UPLATFORM pointer.
UsbPumpPlatform_PostEvent	A synonym for the old UHIL_PostEvent
UsbPumpPlatform_ReleaseEventLock	A synonym for the old UHIL_PostEventLock
UsbPumpPlatform_PostIfNotBusy	Post event if it's not already waiting to be processed
UsbPumpPlatform_MarkCompletionBusy	Mark a completion as waiting to be processed
UsbPumpPlatform_MarkCompletionNotBusy	Mark a completion as eligible for posting
UsbPumpPlatform_SynchronizeToDataPump	Analogous to UHIL_SynchronizeToDataPump, but for platforms.

10.3.1 UsbPumpPlatform_CheckEvent

This function is a renaming of `UHIL_CheckEvent()`, taking a `UPLATFORM` pointer instead of a `UEVENTCONTEXT` pointer. `UHIL_CheckEvent` is retained in the library for compatibility with old code.

```
UCALLBACKCOMPLETION *UsbPumpPlatform_CheckEvent (
   UPLATFORM *pPlatform
   );
```

10.3.2 UsbPumpPlatform_DoEvents

This function is a renaming of `UHIL_DoEvents()`, taking a `UPLATFORM` pointer instead of a `UEVENTCONTEXT` pointer. `UHIL_DoEvents` is retained in the library for compatibility with old code.

```
VOID UsbPumpPlatform_DoEvents(
   UPLATFORM *pPlatform,
```

```
   ULONG nMaxEvents
   );
```

10.3.3 UsbPumpPlatform_DispatchEvent

This function a renaming of `UHIL_dispatchevent()`, taking a `UPLATFORM` pointer instead of a `UEVENTCONTEXT` pointer. `UHIL_dispatchevent` is retained in the library for compatibility with old code.

```
VOID UsbPumpPlatform_DispatchEvent(
   UPLATFORM *pPlatform,
   UCALLBACKCOMPLETION *pCallbackCompletion
   );
```

10.3.4 UsbPumpPlatform_GetEvent

This function is a renaming of `UHIL_GetEvent()`, taking a `UPLATFORM` pointer instead of a `UEVENTCONTEXT` pointer. `UHIL_GetEvent` is retained in the library for compatibility with old code.

```
UCALLBACKCOMPLETION *UsbPumpPlatform_GetEvent(
   UPLATFORM *pPlatform
   );
```

Note that this routine should **not** cause the DataPump thread to block if there are no events available to be processed.

10.3.5 UsbPumpPlatform_GetEventEx

This function is a renaming of `UHIL_GetEventEx()`, taking a `UPLATFORM` pointer instead of a `UEVENTCONTEXT` pointer. `UHIL_GetEventEx` is retained in the library for compatibility with old code.

```
UCALLBACKCOMPLETION *UsbPumpPlatform_GetEventEx(
   UPLATFORM *pPlatform,
   BOOL fWait
   );
```

If `fWait` is `TRUE`, this routine should cause the DataPump thread to block if there are no events available to be processed. Otherwise, if `fWait` is `FALSE`, this routine should return `NULL` if there are no events to be processed.

10.3.6 UsbPumpPlatform_PostEvent

This function is a renaming of `UHIL_PostEvent()`, taking a `UPLATFORM` pointer instead of a `UEVENTCONTEXT` pointer. `UHIL_PostEvent` is retained in the library for compatibility with old code.

```
VOID UsbPumpPlatform_PostEvent(
   UPLATFORM *pPlatform,
   UCALLBACKCOMPLETION *pCompletion
   );
```

A given completion may be posted to the event queue multiple times concurrently, subject to the overall limitation on event queue length.

10.3.7 UsbPumpPlatform_ReleaseEventLock

This function is a renaming of `UHIL_ReleaseEventLock()`, taking a `UPLATFORM` pointer instead of a `UEVENTCONTEXT` pointer. `UHIL_ReleaseEventLock` is retained in the library for compatibility with old code.

```
VOID UsbPumpPlatform_ReleaseEventLock(
   UPLATFORM *pPlatform
   );
```

10.3.8 UsbPumpPlatform_PostIfNotBusy

This function is equivalent to `UsbPostIfNotBusy()`, but takes a `UPLATFORM` pointer as its first argument.

```
VOID UsbPumpPlatform_PostIfNotBusy(
   UPLATFORM *pPlatform,
   UCALLBACKCOMPLETION *pCompletion,
   VOID *pContext
   );
```

If the `UCALLBACKCOMPLETION` is not already marked busy, then it is atomically marked busy, and submitted to the event queue. A given completion may be posted to the event queue at most once until it is marked not-busy. (Normally, the completion's callback function will do this.)

10.3.9 UsbPumpPlatform_MarkCompletionBusy

This function is equivalent to `UsbMarkCompletionBusy()`, but takes a `UPLATFORM` pointer instead of a `UDEVICE` pointer as its first argument.

```
VOID UsbPumpPlatform_MarkCompletionBusy(
   UPLATFORM *pPlatform,
   UCALLBACKCOMPLETION *pCompletion,
   VOID *pContext
   );
```

This routine marks the completion busy, and does so atomically with respect to `UsbPumpPlatform_PostIfNotBusy()`.

10.3.10 UsbPumpPlatform_MarkCompletionNotBusy

This function is equivalent to `UsbMarkCompletionNotBusy()`, but takes a `UPLATFORM` pointer rather than a `UDEVICE` pointer as its first argument.

```
VOID UsbPumpPlatform_MarkCompletionNotBusy(
   UPLATFORM *pPlatform,
   UCALLBACKCOMPLETION *pCompletion
   );
```

This routine marks the completion not-busy, allowing a subsequent call to `UsbPumpPlatform_PostIfNotBusy()` to successfully post the completion to the event queue.

10.3.11 UsbPumpPlatform_SynchronizeToDataPump

This function is analogous to `UHIL_SynchronizeToDataPumpDevice()`. If the DataPump is not busy, then this function will call the function at `pSynchFn` directly and set DataPump busy. If the DataPump is busy, then an event will be posted to the event queue, causing the function to be called at the next opportunity. This routine is often used as linkage between the primary and secondary ISRs of HCDs.

```
VOID UsbPumpPlatform_SynchronizeToDataPump (
   UPLATFORM *pPlatform,
   UPLATFORM_SYNCHRONIZATION_BLOCK *pSynchBlock,
   UPLATFORM_SYNCHRONIZED_FN *pSynchFn,
   VOID *pContext
   );
```

UPLATFORM_SYNCHRONIZATION_BLOCK is an opaque block of memory that is used for handling the communication. UPLATFORM_SYNCHRONIZED_FN has the signature:

```
typedef VOID UPLATFORM_SYNCHRONIZED_FN(
   UPLATFORM *pPlatform,
   VOID *pContext
   );
```

The context pointer is the same pointer that was passed in to `UsbPumpPlatform_SynchronizeToDataPump()` and contains arbitrary information.

Even though the contents of `pSynchBlock` are opaque, the address is passed through to assist code to locate parent objects.

11. MCCI Dynamic Memory Allocation Routines

11.1 Memory Functions In Pre-2.0 DataPump

11.1.1 UsbAllocateDeviceBuffer

Function: Allocates a buffer from the device pool.

Definition:

```
VOID *UsbAllocateDeviceBuffer(
    UDEVICE *pDevice,
    BYTES bufsize
    );
```

Description: USB bus mastering or DMA devices may have address space affinities. To accommodate these without requiring creative (and non-portable) linking and strong linker capabilities, the concept of the device pool is implemented. The intent is for the device pool to be used by peripheral devices that act as bus masters. This module provides limited device pool capabilities -- in particular, it implements a simple pool that has no provisions for "freeing" buffers, or for satisfying alignment constraints.

If hardware needs buffers to be aligned, it is the responsibility of the implementor to replace this routine with a routine that aligns data buffers.

This routine is not intended for use in allocating general data structures; the device pool is intended to be used only for data structures that are to be shared between a bus mastering peripheral and this code.

Returns: Pointer to the data buffer, or NULL.

11.1.2 UsbCreateDevicePool

Function: Creates the device pool from a buffer.

Definition:

```
VOID UsbCreateDevicePool(
    UDEVICE * pDevice,
    VOID *poolbase,
    BYTES poolsize
    );
```

Description: The device pool is initialized to use the memory region specified in the call. The caller is responsible for ensuring all memory allocated from this pool is accessible to the DMA engine of the underlying hardware.

Notes: The point of this routine is to isolate the pool space implementation from the device policy. It is conceivable that we could overlay a better scheme (including buffer-freeing capability) without changing the interface layer. Perhaps it isn't likely!

Returns: Nothing.

11.1.3 UsbPumpLib_DeviceAllocateFromPlatform

Function: Default method for UsbAllocateDeviceBuffer(), and uses the platform malloc function.

Definition:

```
VOID * UsbPumpLib_DeviceAllocateFromPlatform (
    UDEVICE *pDevice,
    BYTES bufsize
    );
```

Description: For devices without device-specific buffer affinity (especially devices that operate using programmed I/O) this routine can be used as the UsbAllocateDeviceBuffer() method. It simply calls the platform memory allocator.

Returns: Pointer to the data buffer, or NULL.

11.1.4 UsbPumpLib_DeviceFreeToPlatform

Function: Default method for freeing buffers allocated with DeviceAllocateFromPlatform().

Definition:

```
VOID UsbPumpLib_DeviceFreeToPlatform (
    UDEVICE *pDevice,
    VOID *pBuffer,
    BYTES nBuffer
    );
```

Description: The buffer is released to the platform. If there is no platform routine, the buffer is leaked.

Returns: No explicit result.

11.1.5 UsbPumpDeviceLib_AllocateAlignedBufferFromPlatform

Function: Allocate aligned memory from the platform memory pool.

Definition:

```
VOID *UsbPumpDeviceLib_AllocateAlignedBufferFromPlatform (
    UDEVICE *   pDevice,
    BYTES       Size,
    BYTES       Alignment
    );
```

Description: This function allocates an aligned memory block from the platform memory pool. If Alignment is less than or equal to equal ALIGNMENT[4], it is the same as the allocated platform memory buffer. Alignment must be power of 2. If not, this function returns NULL.

This allocates Size + Alignment bytes of memory. It makes an aligned memory buffer and saves the adjusted offset value before the aligned buffer. UsbPumpDeviceLib_FreeAlignedBufferToPlatform() will check this information and release the correct buffer.

Returns: Pointer to allocated buffer, or NULL.

11.1.6 UsbPumpDeviceLib_FreeAlignedBufferToPlatform

Function: Free the aligned memory buffer to the platform memory pool.

Definition:

```
VOID *UsbPumpDeviceLib_FreeAlignedBufferToPlatform (
    UDEVICE *   pDevice,
    VOID *      pBuffer,
    BYTES       Size,
    BYTES       Alignment
    );
```

Description: The buffer is released to the platform memory pool. If the buffer is aligned, then updating buffer pointer and buffer size based on "Alignment" and then free it. If not, just free the buffer.

Returns: Nothing.

11.2 Memory Functions In Post-2.0 DataPump

In previous versions of the DataPump, MCCI provided simple platform memory allocation and release routines, available as `UsbPumpPlatform_Malloc()` and `UsbPumpPlatform_Free()`.

With the addition of embedded host support, these routines are somewhat inadequate for our needs.

- The memory pool allocation is not guaranteed to keep track of the allocated size

[4] ALIGNMENT is a constant. It has different values in different compiler. Refer to uhilalign.h for more detail.

- There is no `realloc()`-like primitive

- There is no pre-defined implementation of `free()` – the default implementation uses a fixed size pool and cannot recycle memory once allocated

- There is no way to define fixed or variable sized heaps that are module-specific – this requires excessive (and hard to test) code to explicitly release memory in case of a failure.

We do not wish to break backwards compatibility; but fortunately, there is a relatively simple set of extensions and redefinitions that will allow us to do what we want.

Section 11.2.1 introduces the refined concepts defined for DataPump memory allocation.

Section 11.2.2 discusses redefinitions and additions to the external API, and the platform changes this implies.

11.2.1 Memory Allocation Concepts

Memory is allocated from abstract memory objects known as "pools". The DataPump makes weak assumptions about these pools.

- The DataPump assumes that it is safe to exhaust the memory of a pool – i.e. that the system won't crash if a `NULL` pointer is returned.

- The DataPump assumes that memory blocks allocated from the pool will be correctly aligned for any type.

In previous versions of DataPump, there were additional assumptions and requirements.

- The DataPump product architecture required that it be possible to allocate memory with minimum overhead, apart from that needed to satisfy CPU alignment constraints.

- The DataPump formerly required that, in case of failure, all memory allocated by DataPump code be explicitly released, otherwise memory might leak.

- The DataPump formerly required that the code that was freeing memory be explicitly aware of the size of the memory block being freed.

This is changed in the current version, as follows.

- Memory usage must be minimal, while satisfying other important system requirements listed below.

- The DataPump requires that it be possible for one module to allocate a block of memory from a pool, and for another module subsequently to free that block without knowing which pool the block of memory was allocated from.

- The DataPump requires that it be possible to allocate all the memory for a given subsystem from a given pool, and then to release all the resources as a block. (This is akin to the common distinction between task-local allocation and system-global allocation.) For example, a USB host class driver might have a pool that is used for allocations each time a new instance arrives, but is reset when the instance departs (without requiring explicit memory free operations).

- The DataPump now has a pool that is associated with the `UPLATFORM` object. Memory blocks allocated from this pool are assumed to be persistent for the duration of the DataPump thread.

- This pool might be implemented using underlying facilities provided by the host operating system, or it might be implemented by managing a large array of bytes provided to the DataPump at initialization time. In the latter case, MCCI provides a pool manager that can be used for this purpose.

- The size of a memory block, and the pool containing the memory block, can be determined from the memory block.

- Therefore, the DataPump (implicitly) has class "AllocatedMemoryBlock" containing the methods `p->Size()` and `p->Heap()`. The methods `p->Heap()->Free()are further defined`, which will release the object p without needing a-priori knowledge of which heap the block was allocated from.

11.2.2 Memory Allocation API Changes

Most existing platforms for the DataPump do not provide a system-wide allocation routine. Therefore, the platform memory allocation routines are redefined to interoperate with the abstract pool mechanisms, and a new field, `upf_pPoolHeader` is added to the `UPLATFORM`:

```
USBPUMP_ABSTRACT_POOL *upf_pPlatformPool;
```

To access the Abstract Pool and Memory Block functions, the following header file should be included:

```
#include "usbpump_mempool.h"
```

11.2.2.1 UsbPumpPlatform_Malloc

This function (along with its derivative, `UsbPumpPlatform_MallocZero`) has the same API as in previous releases, but with additional guarantees. These routines are now required to be based on an abstract pool. This means that any block allocated by `UsbPumpPlatform_Malloc` may be freed using `UsbPumpMemoryBlock_Free()`.

The platform functions `upf_pMalloc`, etc., are treated differently in this release. For platforms that have not been converted to the new allocation scheme, there is a "default" abstract pool that

will call the existing function pointers. For platforms that have been converted, a different abstract pool methodology can be substituted as needed; and the new methodology need not use the `upf_pMalloc` pointers. This means there will be changes in the `UPLATFORM_SETUP_Vx` macros, and that platforms will need to be converted to take advantage of future enhancements in the `UPLATFORM` api.

11.2.2.2 UsbPumpPlatform_Free

This function still accepts a size parameter as input, but the size is no longer used. Instead, it calls `UsbPumpMemoryBlock_Free()`.

An abstract pool header follows the general "union/struct" hierarchy used by DataPump abstract types. The fully abstract type has the following fields:

`VOID *AbstractPool.pContext;`	Context pointer for use by the implementation
`USBPUMP_POOL_ALLOC_FN *AbstractPool.pAlloc;`	Allocation function
`USBPUMP_POOL_BLOCK_REALLOC_FN *AbstractPool.pRealloc;`	Re-sizing function
`USBPUMP_POOL_BLOCK_FREE_FN *AbstractPool.pFree;`	Function for releasing a block.
`USBPUMP_POOL_RESET_FN *AbstractPool.pReset;`	Function for resetting the pool.
`USBPUMP_POOL_CLOSE_FN *AbstractPool.pClose;`	Function for closing the pool header prior to deleting it.

As usual, to facilitate creating a derived type, two types are defined:

1. The abstract pool object itself is a union, containing a single view:

   ```
   USBPUMP_ABSTRACT_POOL_CONTENTS   AbstractPool;
   ```

2. The abstract pool contents structure in turn contains the fields listed above.

   `USBPUMP_ABSTRACT_POOL_CONTENTS__UNION` should be used as the union prefix when creating a union derived from `USBPUMP_ABSTRACT_POOL`.

`USBPUMP_ABSTRACT_POOL_CONTENTS__STRUCT` should be used as structure prefix when defining a structure derived from an abstract pool

The method functions have the following definitions:

```
typedef VOID *USBPUMP_POOL_ALLOC_FN(
   USBPUMP_ABSTRACT_POOL *pHeader,
```

```
      ADDRBITS nBytes
      );

typedef VOID *USBPUMP_POOL_REALLOC_FN(
      USBPUMP_ABSTRACT_POOL *pHeader,
      VOID *pBlock,
      ADDRBITS nBytes
      );

typedef VOID USBPUMP_POOL_FREE_FN(
      USBPUMP_ABSTRACT_POOL *pHeader,
      VOID *pBlock
      );

typedef VOID USBPUMP_POOL_RESET_FN(
      USBPUMP_ABSTRACT_POOL *pHeader
      );

typedef VOID USBPUMP_POOL_CLOSE_FN(
      USBPUMP_ABSTRACT_POOL *pHeader
      );
```

To realloc a block, call:

```
VOID *UsbPumpMemoryBlock_Realloc(VOID *pMemoryBlock, ADDRBITS NewSize);
```

This function works exactly as C89's `realloc()` function, except that if `pMemoryBlock` is `NULL` the result is always `NULL`.

For a function with full C89 semantics, use

```
VOID *UsbPumpPool_Realloc(USBPUMP_ABSTRACT_POOL *pPool, VOID *pMemoryBlock,
ADDRBITS NewSize);
```

If `pMemoryBlock` is `NULL`, then `NewSize` bytes are allocated from `pPool`. Otherwise, the memory block is resized within its existing pool (and `pPool` is not used).

To free a block, call:

```
VOID UsbPumpMemoryBlock_Free(VOID *pMemoryBlock);
```

`UsbPumpMemoryBlock_Free` checks whether `pMemoryBlock` is `NULL`; if not, it uses `USBPUMP_ABSTRACT_POOL_BLOCK_TO_HEADER` to find the abstract pool object, and calls the `AbstractPool.pFree` function.

To locate the Abstract Pool Object given a memory block allocated from any pool, use:

```
USBPUMP_ABSTRACT_POOL *
USBPUMP_MEMORY_BLOCK_GET_ABSTRACT_POOL(
            VOID *pMemoryBlock
            );
```

`pMemoryBlock` must not be `NULL`.

A safe version of the macro that will not dereference a NULL pointer (but which will possibly return a NULL pointer if the input pointer is NULL) is:

```
USBPUMP_ABSTRACT_POOL *
UsbPumpMemoryBlock_GetAbstractPool(
          VOID *pMemoryBlock
          );
```

As an implementation note, normally the Pool Header pointer is found at `((ADDRBITS*)pBlock)[-1]`. This yields the abstract pool header, which then can be cast to a pointer and used to free the block.

11.2.2.3 UsbPumpPlatform_CreateAbstractPool

This function creates an abstract memory pool using the platform method. The platform's upf_pCreateAbstractPool function is called to create an abstract memory pool of the specified size. If not this function itself allocate memory and create abstract memory pool. If memory not available, it returns NULL.

```
USBPUMP_ABSTRACT_POOL *
UsbPumpPlatform_CreateAbstractPool(
   UPLATFORM * pPlatform,
   BYTES       SizeMemoryPool
   );
```

11.2.3 The default DataPump memory allocation package

A linear pool control object has the following layout:

```
USBPUMP_ABSTRACT_POOL_CONTENTS
AbstractPool;

VOID *LinearPool.pBase;                 Base address of the buffer

ADDRBITS LinearPool.nBytesInUse;        Number of bytes in use (raw)

ADDRBITS LinearPool.nBytesTotal;        Number of bytes total in the pool
```

Each allocated memory object starts with the following header:

```
ADDRBITS PoolHeader[2];
```

Normally the pool header is defined as:

```
ADDRBITS Size;          Size of the block, in bytes, including waste space and the header.
```

```
ADDRBITS PoolHeader;       Base address of the pool-header, as a UINT_PTR.
```

The minimum size of a pool block (including the header) is twice the header size, and the granularity of allocation is the header size.

To create a linear pool, call:

```
VOID UsbPumpLinearPool_Initialize(
   USBPUMP_LINEAR_POOL *pPoolHeader,
   VOID *pPoolBuffer,
   ADDRBITS SizePoolBuffer
   );
```

To create a linear pool within a buffer, call:

```
USBPUMP_LINEAR_POOL *UsbPumpLinearPool_Create(
   VOID *pPoolBuffer,
   ADDRBITS SizePoolBuffer
   );
```

This routine first carves a `USBPUMP_LINEAR_POOL` header out of the specified buffer, and then initializes it to describe the rest of the buffer. The first few bytes of the buffer are discarded, if needed to satisfy alignment constraints.

12. MCCI USB DataPump Internal APIs

12.1 Initialization

The initialization of the DataPump device stack is table-driven. The table interpreter is UsbPump_GenericApplicationInit (). It processes an application initialization vector which comes from configuration data or somewhere else. The MCCI DataPump, uses a standard name, gk_UsbPumpApplicationInitHdr. Different builds might have different files that define different versions of gk_UsbPumpApplicationInitHdr.

The GenericApplicationInit () function looks at the descriptors from the URC file and compares them to the info in the application init vector. It *conditionally* creates protocols if it finds the descriptor set actually can support a protocol that has been configured into the Application Init Vector. This decouples the behavior of the overall device from the descriptors, which is very useful for test, for maintenance, and for using a single code base across a family of products.

It requires the system engineer be selective with what is included in the application init vector. The app init vector affects the ROM footprint directly. The URC file combined with the app init vector affects the RAM footprint, based on which protocols are instantiated.

12.1.1 App Init Header

Look at usbkern/apps/wmcdemo/mscacmdemo/mscacmdemo_appinit.c.

```
CONST USB_DATAPUMP_APPLICATION_INIT_VECTOR_HDR gk_UsbPumpApplicationInitHdr =
   USB_DATAPUMP_APPLICATION_INIT_VECTOR_HDR_INIT_V1(
       UsbPumpApplicationInitVector,
       /* pSetup */ NULL,
       /* pFinish */ MscDemoI_AppInit_VectorFinish
       );
```

This header provides the vector of application initialization nodes and a pointer to a function to be called after initialization is complete. This function is specific. One UDEVICE is created for each app init node element that "matches".

12.1.2 Proto Init Header

Still use MSC as an example.

```
static
CONST USBPUMP_PROTOCOL_INIT_NODE_VECTOR InitHeader =
   USBPUMP_PROTOCOL_INIT_NODE_VECTOR_INIT_V1(
       /* name of the vector */ InitNodes,
       /* prefunction */ NULL,
       /* postfunction */ NULL
       );
```

The main purpose of this header is to link to the InitNodes. The init macro fills in the size of the vector in the init header. Prefunction and postfunction are hooks for doing special things, and are not normally used.

USBPUMP_PROTOCOL_INIT_NODE_INIT_V2 or
USBPUMP_PROTOCOL_INIT_NODE_INIT_V1 macros
are used to initialize USBPUMP_PROTOCOL_INIT_NODE. Each proto init node has some standard matching fields:

- Normally match based on bInterfaceClass, bInterfaceSubClass and bInterfaceProtocol from the interface descriptor
- Can also qualify based on bDeviceClass etc.
- The protocol implementation can supply a probe function that further qualifies the match.
- All of the above can be "wild-carded."

The following provides an explanation of MSC protocol init node:

```
static CONST USBPUMP_PROTOCOL_INIT_NODE InitNodes[] =
   {
   USBPUMP_PROTOCOL_INIT_NODE_INIT_V2(
```

```
            /* dev class, subclass, proto */ -1, -1, -1,
            /* ifc class */ USB_bInterfaceClass_MassStorage,
            /* subclass */ USB_bInterfaceSubClass_MassStorageScsi,
            /* proto */ -1,
            /* cfg, ifc, altset */ -1, -1, -1,
            /* speed */ -1,
            /* uProbeFlags */ USBPUMP_PROTOCOL_INIT_FLAG_AUTO_ADD,
            /* probe */ NULL,
            /* create */ MscSubClass_Atapi_ProtocolCreate,
            /* pQualifyAddInterface */ NULL,
            /* pAddInterface */ NULL,
            /* optional info */ &gk_MscDemoI_RamdiskConfig
            )
        };
```

- –1 is generally a wild card, rendering dev class, subclass, proto, etc. irrelevant.
- USB_bInterfaceClass_MassStorage constants originate from the mass-storage header file usbmsc10.h.
- cfg, ifc, altset allow a protocol node to match structurally.
- MscSubClass_Atapi_ProtocolCreate is provided by the protocol implementation, and it is the primary entry point. All other code is included automatically by the linker.
- gk_MscDemoI_RamdiskConfig is used by the protocol implementation to configure how it operates – the protocol manual should specify use.

Suppose we have the following entries:

- An entry that matches mass storage
- An entry that matches ACM modems
- An entry that matches anything that hasn't been matched

The initialization framework first looks at every UINTERFACE in the descriptor set, and sees if it is mass storage. If so, a new mass storage instance is created for that interface, and that interface is marked "in use". The framework repeats from the beginning, but this time looks for ACM modems. Finally, the framework repeats from the beginning and creates an instance of LOOPBACK for each UINTERFACE that isn't already "in use".

12.1.3 Port Init Header

The Port Init Header configures support for the USB device hardware. Port initialization is similar to the BSP concept in embedded system. This is for binding DCDs. The list of possible DCDs is determined by the USB_DATAPUMP_PORT_INIT_VECTOR. Each vector contains one or more USB_DATAPUMP_PORT_INIT_VECTOR elements.

```
USB_DATAPUMP_PORT_INIT_VECTOR_INIT_V1(
        /* optional probe fn */ pPortProbeFunction,
        /* DCD primary export: table of functions */ pDeviceSwitch,
        /* bus handle to be used by DCD for this DC */ hBus,
        /* base I/O address on bus for this DC */ ioPort,
        /* "wiring information" tailor DCD to this hw */ pConfigInfo,
        /* size of info, for reference */ sizeConfigInfo
        )
```

The name of the device switch exported by the DCD normally is defined in the DCD header file.

hBus and ioPort specify the base address and bus handle used by the DCD for accessing the registers of the device. Often, hBus is not used, but it is architecturally required.

pConfiginfo is used to tell the DCD about how the USB interface is wired up. The format of this structure is defined by the DCD code, so you must refer to the header files or to the DCD manual to find out more about this.

If there are multiple ports, e.g., wired USB device controller and a wireless USB device controller, the probe function in the Port Init vector is used to find ports. Each time a port probe succeeds, the application init vector is scanned for that port (according to the rules given previously). When it's time to initialize a DCD instance, the UDEVSWITCH pointer from the port vector is used (but in the context of the code from the URC file)

12.2 Configuration Management Functions (Internal Use Only)

The functions defined in this section are for use internally to the DataPump core logic. They are documented here for reference.

12.2.1 UsbChangeConfig

Function: Select a new configuration (obsolete).

Definition:

```
#include "usbpump.h"

BOOL UsbChangeConfig(
            UDEVICE *self,
            UCONFIG *pNewConfig,
            UEVENT whyin,
            UEVENT whyout,
            VOID *info
            );
```

Description: This routine is an obsolete API, provided for backwards compatibility. It calls UsbChangeConfigEx() with the same parameters, and ignores the result.

Returns: No explicit result.

12.2.2 UsbChangeConfigEx

Function: Select a new configuration, and return status.

Definition:

```
#include "usbpump.h"

BOOL UsbChangeConfig(
            UDEVICE *self,
            UCONFIG *pNewConfig,
            UEVENT whyin,
            UEVENT whyout,
            VOID *info
            );
```

Description: This routine is called to change the device's configuration state. If `pNewConfig` is NULL, then the new state will be "unconfigured"; if non-NULL, then the new state will be "configured", with the configuration selected by the pointer.

If the configuration is not changing, this is a no-op.

If the device is already configured, the old configuration is torn down, stopping all I/Os, and disconnecting the pipes.

If the device is to go to the "configured" state, a new configuration is set up.. Because the default pipe is not reachable from a properly constructed data structure, the default pipe is not affected by this operation.

The old configuration is notified before tear down; the new configuration is notified after set up. There is no NULL configuration, so if the device is going to/from unconfigured, only the real configurations get notified. However, the device gets notified for configuration changes.

Because the interfaces/configs tend to want to queue I/O requests, the device is notified to change the hardware setting after the data structures are changed but before the interface is notified.

Returns: No explicit result.

12.2.3 UsbChangeInterface

Function: Make a particular alternate setting the active alternate setting for its interface.

Definition:

```
#include "usbpump.h"

VOID UsbChangeInterface(
            UDEVICE *self,
            UINTERFACE *pifc,   -- the interface to make current
```

```
    UEVENT whyin,       -- event code to give new ifc,
    UEVENT whyout,      -- event code to give old ifc,
    VOID *info          -- info for both.
    );
```

Description: If the "alt interface" is changing, the old alt interface is notified, the I/O is killed on all the endpoints and associations are torn down. The new alt interface is then set up and notified that it has been set up.

Because the ifcset is notified before any changes and after all changes, the ifcset can do pre/post with a common function. Applications can use that hook to do their work.

Returns: Nothing.

12.2.4 UsbInterfaceSetup

Function: Make a specific interface the "current" interface of its set.

Definition:

```
VOID UsbInterfaceSetup (
    UDEVICE *self,
    UINTERFACE *pnewifc
    );
```

Description: The "current" interface is established in the normal way: all of its pipes are bound to their associated endpoints. The various event queues are notified.

Returns: Nothing.

Notes: Someone else must have torn down the previous interface setting, if needed.

Someone else must notify the interfaces. We don't do it here, because we want all the data structure changes made, and then hardware structures changed, before interfaces are notified.

12.2.5 UsbInterfaceSetupNotify

Function: Tell a specific interface that it's now the "current" interface of its set.

Definition:

```
#include "usbpump.h"

   VOID UsbInterfaceSetupNotify (
       UDEVICE *self,
       UINTERFACE *pnewifc
       UEVENT whyin,
```

```
        VOID *info
        );
```

Description: The various event queues are notified.

Returns: Nothing.

Notes: Someone else must have set up the interface.

12.2.6 UsbInterfacesetTeardown

Function: Tear down the current set of pipes associated with a given interface set.

Definition:

```
#include "usbpump.h"

   VOID UsbInterfacesetTeardown (
       UDEVICE *self,
       UINTERFACESET *piset
       );
```

Description: The "current" interface of the specified interface-set is deconfigured. Deconfigure means: changing all endpoints such that any endpoint referring to one of the pipes in the interface is disconnected from the interface, and associated I/O is cancelled.

Returns: Nothing.

12.2.7 UsbInterfacesetTeardownNotify

Function: Notify the current set of pipes associated with a given interface set that they are about to be torn down.

Definition:

```
#include "usbpump.h"

   VOID UsbInterfacesetTeardownNotify (
       UDEVICE *self,
       UINTERFACESET *piset,
       UEVENT whyout,
       VOID *info
       );
```

Description: The "current" interface of the specified interface-set is notified that it is about to be deconfigured: notifying the interface-set, the interfaces, and all the pipes of what is about to happen.

Returns: Nothing.

12.2.8 UsbInterfacesetTeardownNotify

Function: Notify the current set of pipes associated with a given interface set that they are about to be torn down.

Definition:

```
#include "usbpump.h"

   VOID UsbInterfacesetTeardownNotify (
       UDEVICE *self,
       UINTERFACESET *piset,
       UEVENT whyout,
       VOID *info
       );
```

Description: The "current" interface of the specified interface-set is notified that it is about to be deconfigured: notifying the interface-set, the interfaces, and all the pipes of what is about to happen.

Returns: Nothing.

12.2.9 UsbInterfaceActivateEndpoints

Function: Activate or deactivate all endpoints associated with an interface.

Definition:

```
#include "usbpump.h"

   VOID UsbInterfaceActivateEndpoints( (
       UDEVICE *   pDevice,
       UINTERFACE *pInterface,
       BOOL    fActive
       );
```

Description: Walk thru all pipes associated with this dataplane and activate or deactivate the endpoints in each interface.

Returns: Nothing.

12.3 Device Related Functions

12.3.1 UsbPumpDevice_CheckAutoRemoteWakeup

Function: Check auto remote wakeup and complete the remote wakeup process.

Definition:

```
#include "udevice.h"

        BOOL   UsbPumpDevice_CheckAutoRemoteWakeup(
               UDEVICE      *      pDevice,
               UINTERFACE *  pInterface
               );
```

Description: This routine checks automatic device or function remote wakeup state. If the current device speed is not SuperSpeed, this routine checks device remote wakeup. Except in SuperSpeed, function remote wakeup is not possible in USB, so in this case pInterface is not used.

If the current device speed is SuperSpeed, this routine instead will check function remote-wakeup state for the function containing the interface selected by pInterface. If pInterface is NULL, this function checks all UINTERFACEs in the current configuration.

In either case, if device or function remote wakeup is enabled, remote wakeup is activated.

Returns: TRUE if remote wakeup is required, otherwise FALSE.

12.3.2 UsbPumpDevice_AllocateDeviceBuffer

Function: Allocate a buffer from the device pool.

Definition:

```
#include “udevice.h"

        VOID UsbPumpDevice_AllocateDeviceBuffer(
                UDEVICE *    pDevice,
                BYTES        nBytes
                );
```

Description: This function allocates a buffer from the device pool. If there is no device pool, it will allocate a buffer from the platform pool. Clients use this routine to allocate a buffer that is guaranteed to be acceptable for DMA purposes.

Device pool is used for devices with limited DMA capabilities. If a device can DMA anywhere is system memory without restriction, then device pool is not normally used.

Returns: Pointer to the data buffer, or NULL if a buffer cannot be allocated.

Note: Buffers allocated using UsbPumpDevice_AllocateDeviceBuffer() must be freed using UsbPumpDevice_FreeDeviceBuffer().

12.3.3 UsbPumpDevice_FreeDeviceBuffer

Function: Return a buffer to the device pool.

Definition:

```
#include "udevice.h"

VOID UsbPumpDevice_FreeDeviceBuffer(
        UDEVICE *   pDevice,
        VOID *      pBuffer,
        BYTES       nBytes
        );
```

Description: This function releases a buffer allocated by UsbPumpDevice_AllocateDeviceBuffer(). nBytes must be the same value that was used to allocate the buffer.

Returns: None

12.3.4 UsbPumpDeviceI_SetLinkState

Function: Set USB3 link state.

Definition:

```
#include "udevice.h"

BOOL   UsbPumpDeviceI_SetLinkState(
       UDEVICE      *      pDevice,
       UDEVICE_LINK_STATE  LinkState
       );
```

Description: Set USB3 link state.

Returns: No explicit result.

12.3.5 UsbPumpDevice_GetMaxControlMaxPacketSize

Function: Get maximum wMaxPacketSize for control endpoint.

Definition:

```
#include "udevice.h"

        BOOL   UsbPumpDevice_GetMaxControlMaxPacketSize(
               UDEVICE       *      pDevice
               );
```

Description: This function checks control endpoint max packet size for all supported speeds and returns the maximum value of the control endpoint's max packet size.

Returns: Max packet size of control endpoint.

12.3.6 UsbPumpDevice_QueryBulkIntInPendingQe

Function: Query device for the status of queue element of Bulk/Int IN pipe.

Definition:

```
#include "udevice.h"

        BOOL   UsbPumpDevice_QueryBulkIntInPendingQe(
               UDEVICE       *      pDevice
               );
```

Description: The routine is called to check the status of queue element of Bulk/Int IN pipe. If the "current" configuration is not set, there will be no queue element filled. Otherwise, the active pipes are traversed and return TRUE if there is queue element scheduled.

Returns: TRUE if there is queue element filled in Bulk/Int IN piep, FALSE otherwise.

12.3.7 UsbPumpDevice_QueryRemoteWakeupTrafficPending

Function: Query the status of the remote wakeup pending traffic.

Definition:

```
#include "udevice.h"

        BOOL   UsbPumpDevice_QueryRemoteWakeupTrafficPending(
               UDEVICE       *      pDevice
               );
```

Description: The routine is called to check the status of remote wakeup pending traffic. If the "current" configuration is not set, there will be no queue element filled. Otherwise the active pipes are traversed and return TRUE if there is queue element scheduled.

Returns: TRUE if there is queue element filled in Bulk/Int IN pipe, FALSE otherwise.

12.3.8 UsbPumpDevice_QueryInterfaceRemoteWakeupTrafficPending

Function: Query the status of the remote wakeup pending traffic for UINTERFACE.

Definition:

```
#include "udevice.h"

BOOL   UsbPumpDevice_QueryInterfaceRemoteWakeupTrafficPending(
       UDEVICE      *     pDevice,
       UINTERFACE   *     pInterface
       );
```

Description: The routine is called to check the status of remote wakeup pending traffic for specified pInterface. It will traverse the active pipes and return TRUE if there is queue element scheduled.

Returns: TRUE if there is queue element filled in Bulk/Int IN pipe, FALSE otherwise.

12.3.9 UsbPumpDevice_SuperSpeedFeatureSetup

Function: Set up SuperSpeed device feature.

Definition:

```
#include "udevice.h"

BOOL   UsbPumpDevice_SuperSpeedFeatureSetup(
       UDEVICE      *     pDevice,
       );
```

Description: This routine checks sets up the function wake feature mechanism if given device supports SuperSpeed operation.

Returns: No explicit result.

12.3.10 UsbPumpDeviceI_Delay

Function: Request platform to support waiting when the timer is not started.

Definition:

```
#include "udevice.h"

      VOID * UsbPumpDeviceI_Delay(
            UDEVICE *    pDevice,
            USBPUMP_MILLISECONDS      uMilliseconds
            );
```

Description: When there is no support for device timer, an IOCTL is sent to request waiting.

Returns: None.

12.4 Event Support Functions

12.4.1 UsbReportDeviceEvent

Function: Notifies listeners of state changes.

Definition:

```
#include "usbpumpapi.h"

   VOID UsbReportDeviceEvent (
       UDEVICE *self,
       UEVENT event,
       VOID *pInfo
       );
```

Description: This routine is called when a state change occurs that may require hardware updates. It calls UDEVREPORTEVENT() and UsbReportEvent(self, self->udev_noteq, ...) to pass the event to the hardware layer and to the global application notification queues.

This routine is for internal use by DataPump code only. DCDs should use UsbProcessReset() and so forth, rather than calling this routine directly.

Returns: Nothing.

12.4.2 UsbPumpDevice_SendFunctionWake

Function: Send function wake device notification.

Definition:

```
#include "udevice.h"

      BOOL   UsbPumpDevice_SendFunctionWake(
            UDEVICE      *      pDevice,
            UINTERFACE   *      pInterface
            );
```

Description: This routine sends function wake device notification to the DCD using UEVENT_DEVICE_NOTIFICATION.

Returns: No explicit result.

12.4.3 UsbPumpDevice_DeviceFsm_evDetach

Function: Notify the device fsm of hardware-level Vbus detach.

Definition:

```
#include   "usbpump_device_fsm.h"

   VOID    UsbPumpDevice_DeviceFsm_evDetach(
           UDEVICE *   pDevice
           );
```

Description: Clear fVBus flag and call the fsm evaluation.

Returns: No explicit result.

12.4.4 UsbPumpDevice_DeviceFsm_evAttach

Function: Notify the device fsm of hardware-level Vbus detach.

Definition:

```
#include   "usbpump_device_fsm.h"

   VOID    UsbPumpDevice_DeviceFsm_evAttach(
           UDEVICE *   pDevice
           );
```

Description: Set fVBus flag and call the fsm evaluation.

Returns: No explicit result.

12.4.5 UsbPumpDevice_DeviceFsm_evReset

Function: Notify the device fsm of hardware-level Vbus detach.

Definition:

```
#include   "usbpump_device_fsm.h"

   VOID    UsbPumpDevice_DeviceFsm_evReset(
           UDEVICE *   pDevice,
```

```
                USBPUMP_DEVICE_SPEED newSpeed
                );
```

Description: Set fReset flag and call the fsm evaluation function.

Returns: No explicit result.

12.4.6 UsbPumpDevice_DeviceFsm_evSetAddress

Function: Notify the device fsm of SetAddress request.

Definition:

```
    #include    "usbpump_device_fsm.h"

       VOID     UsbPumpDevice_DeviceFsm_evSetAddress(
                UDEVICE *    pDevice,
                CONST USETUP *   pSetup,
                UCHAR *      pSetupPacket
                );
```

Description: Update the fAddressed flag and call the fsm evaluation function.

Returns: No explicit result.

12.4.7 UsbPumpDevice_DeviceFsm_evSetConfig

Function: Notify the device fsm of configuration change.

Definition:

```
    #include    "usbpump_device_fsm.h"

       BOOL     UsbPumpDevice_DeviceFsm_evSetConfig(
                UDEVICE *    pDevice,
                ARG_USHORT   config,
                UINT8 *      pSetupPacket
                );
```

Description: Set the fCfgChange flag and call the fsm evaluation function.

Returns: TRUE if the configuration was accepted, FALSE otherwise.

12.4.8 UsbPumpDevice_DeviceFsm_evSuspend

Function: Notify the device fsm of hardware-level suspend.

Definition:

```
#include    "usbpump_device_fsm.h"

    VOID    UsbPumpDevice_DeviceFsm_evSuspend(
            UDEVICE *   pDevice,
            );
```

Description: Set the fSuspended flag and call fsm evaluation function.

Returns: TRUE if the configuration was accepted, FALSE otherwise.

13. MCCI USB DataPump API

13.1 Queuing and Completing Requests

13.1.1 UsbGetQe

Function: Get first waiting UBUFQE from specified list.

Definition:

```
PUBUFQE UsbGetQe (
    PUBUFQE *head
    );
```

Description: Take the next QE from list and update list head as required.

Returns: Pointer to next UBUFQE or NULL if list is empty.

13.1.2 UsbPutQe

Function: Add a UBUFQE to the tail of the given list.

Definition:

```
VOID UsbPutQe (
    UBUFQE **pphead,
    UBUFQE *pqe
    );
```

Description: Add UBUFQE to list tail, update head pointer if list is empty.

Returns: Nothing.

13.1.3 UsbCompleteQE

Function: Complete a queue element that is in a list.

Definition:

```
BOOL UsbCompleteQE (
    UDEVICE *pSelf,
    UBUFQE  **pHead OPTIONAL,
    UBUFQE  *pThis,
    USTAT Status
    );
```

Description: Remove the queue element from the list it is on, and complete it. The head pointer is updated, if necessary.

If the UBUFQE's next pointer is NULL, it is completed, but it is assumed it is not on any list.

Returns: TRUE if and only if pHead was set to a non-null value; FALSE if pThis was NULL, pHead was NULL, or *pHead was set to NULL after completing the queue element (which was therefore the last one on the list).

Example: To cancel all the ubufqes in a given list, use:

while (UsbCompletQE(pDevice, &qhead, qhead, USTAT_KILL))

    /* do nothing */;

Note that protocols should not use this routine on UBUFQEs that have been passed to the primitive queuing function; it is intended as a convenience for protocols that maintain their own queues.

13.1.4 UsbCompleteQEList

Function: Complete each UBUFQE in a list of UBUFQEs.

Definition:

```
VOID UsbCompleteQEList(
    UDEVICE *pDevice,
    UBUFQE *pQeHead,
    USTAT Status
    );
```

Description: This routine walks a list of UBUFQEs, completing each one in turn. It assumes that each element is initialized to the point that it is safe to call UsbCompleteQE() on it.

Returns: No explicit result.

13.1.5 UsbEndpointCancelIo

Function: Cancel (and complete) all pending I/Os for a given endpoint.

Definition:

```
VOID UsbEndpointCancelIo (
    UDEVICE *udev,
    UENDPOINT *uep,
    USTAT why
    );
```

Description: Any active I/O operations for the endpoint are cancelled; and all pending I/O operations are completed.

Returns: Nothing.

13.1.6 UsbPipeQueue

Function: Queue a buffer-queue-element to a specific USB endpoint.

Definition:

```
VOID UsbPipeQueue (
    UDEVICE *self,
    UBUFQE  *pBufqe
    );
```

Description: The specified buffer is enqueued for the data stream associated with the particular pipe. The type of operation initiated is implied by the endpoint type. If enqueued for a CONTROL_IN endpoint, then the buffer will be sent to the host as part of the current transaction. If enqueued for a CONTROL_OUT endpoint, then the buffer will be filled from the host as part of the current transaction.

After enqueuing, the I/O on the endpoint is started.

Returns: Nothing.

Notes: Maybe should check endpoint type and return errors. Maybe should provide data aggregation features. Fractional packets, packet vs. buffer boundaries, and status synchronization are handled using the UBUFQEFLAG_xxx flags.

See also: UsbPipeQueueBuffer (), which also initializes the donefn and done info.

13.1.7 UsbPumpPipe_QueueList

Function: Queue a single list of UBUFQEs to a pipe.

Definition:

```
VOID UsbPumpPipe_QueueList (
    UDEVICE *pDevice,
    UBUFQE *pListHead,
    BOOL fNeedInit
    );
```

Description: This routine is similar to UsbPipeQueue (), except that it enqueues a list of one or more UBUFQEs to the endpoint given by the first UBUFQE in the list.

Code should have prepared the following entries in each UBUFQE:

1. UPREPIO() on the buffer.
2. bufindex = 0
3. bufars = 0
4. pPipe = the pipe pointer.

If unsure about this initialization, set fNeedInit TRUE, and all the initialization will be performed.

Returns: Nothing.

See also: UsbPipeQueue ().

13.1.8 UsbPipeQueueBufferWithTimeout

Function: Queue a buffer, setting the timeout if the flags so indicate.

Definition:

```
VOID UsbPipeQueueBufferWithTimeout(
    PUDEVICE self,
    PUPIPE  pPipe,
    PUBUFQE pqe,
    VOID*buf,
    BYTES buflen,
    UBUFIODONEFN *pdonefn,
    VOID*doneinfo,
    UBUFQE_FLAGS flags,
    UBUFQE_TIMEOUT timeout
    );
```

Description: Some callers may want to specify the timeout for completing a buffer if no short packet is received, because it is not constant for a given packet; the same callers are likely to want to set the buffer pointers in the UBUFQE, rather than reuse from before.

This routine encapsulates that operation: the completion function and info are stored into the UBUFQE, and then UsbPipeQueue () is called.

If the value of flags does not have UBUFQEFLAG_IDLE_COMPLETES, then timeout is not used. If timeout is zero, no timeout is used.

Returns: Nothing.

Notes: If additional fields are added to the UBUFQE, then this routine is required to initialize them in such a way that old code remains compatible.

The UBUFQEFLAG_IDLE_COMPLETES option requires an idle value.

If just the DataPump default timeout is desired, you can use:

UsbPumpQueueBuffer (), where a pre-defined timeout is used.

See also: UsbPumpQueueBuffer (), UsbPipeQueue ()

13.1.9 UsbPipeQueueBuffer

Function: This convenience routine sets completion information in a UBUFQE, then calls UsbPipeQueue.

Definition:

```
VOID UsbPipeQueueBuffer (
    UDEVICE     *self
    UPIPE       *pPipe,
    UBUFQE      *pBufqe,
    VOID        *pBuf,
    USHORT      size_buf,
    UBUFIODONEFN *pDonefn,
    VOID        *pDoneInfo,
    UBUFQE_FLAGS flags
    );
```

Description: Some callers may want to specify the completion function when a packet is queued, because it is not constant for a given packet; The same callers are likely to want to update the buffer pointers.

This routine centralizes that operation: the completion function and info are stored into the UBUFQE, and then UsbPipeQueue () is called.

Returns: Nothing.

Notes: If additional fields are added to the UBUFQE, then this routine is required to initialize them in such a way that old code remains compatible.

The UBUFQEFLAG_IDLE_COMPLETES option results in a default timeout value of <duration> <units>. Use UsbPumpQueueBufferWithTimeout (), to specify a different timeout duration.

13.2 Descriptor functions

13.2.1 UsbFindInterfaceDescriptor

Function: Locate the interface descriptor that goes with a given USB UINTERFACE entry.

Definition:

```
CONST USBIF_IFCDESC_WIRE *UsbFindInterfaceDescriptor (
        CONST UDEVICE    *pDevice,
        CONST UINTERFACE *pInterface
        );
```

Description: The interface descriptor is located in the constant descriptors produced by USBRC for this interface.

Returns: Pointer to interface; or NULL if an error has occurred.

13.2.2 UsbFindConfigurationDescriptor

Function: Locate the configuration descriptor that goes with a given configuration descriptor index.

Definition:

```
CONST USBIF_CFGDESC_WIRE *UsbFindConfigurationDescriptor(
        CONST UDEVICE    *pDevice,
        UINT8             iCfg
        );
```

Description: The configuration descriptor is located in the constant descriptors produced by USBRC.

Returns: Pointer to the configuration descriptor; or NULL if an error has occurred

13.2.3 UsbFindIndexForConfiguration

Function: Return the configuration index (set-config number)for a given UCONFIG.

Definition:

```
INT UsbFindIndexForConfiguration(
        CONST UCONFIG   *pConfig
        );
```

Description: Find the configuration index by examining the relative position of the input UCONFIG in the containing UDEVICE"s device tree.

Returns: Configuration number (-1 if this configuration is not really part of the parent UDEVICE). Note that 0 is not part of the range of this function.

13.2.4 UsbFindIndexForInterface

Function: Find the alternate setting number for a given UINTERFACE.

Definition:

```
INT UsbFindIndexForInterface (
        CONST UINTERFACE    *pIfc
        );
```

Description: If the DataPump is configured for variable size UINTERFACEs, the table is scanned through to find the match; otherwise it is worked backwards from the position of the ifc in the parent UINTERFACESET's list of alt settings.

Returns: The alternate setting number, or -1 if not defined.

13.2.5 UsbFindIndexForIfcset

Function: Get the index (address) for the given UIFCSET.

Definition:

```
BOOL  UsbFindIndexForIfcset(
        CONST UDEVCE *self,
        UIFCSET *ifcset,
        USHORT *pindex
        );
```

Description: Searches through the interfaces for the current configuration until a match is found or the search is exhausted,

Returns: TRUE: `ifcset` found; index saved in `*pindex`.

FALSE: `ifcset` not found. `*pindex` is not valid.

13.2.6 UsbFindIndexForInterfaceSet

Function: Get the index (interface number) for the given Ifcset.

Definition:

```
INT UsbFindIndexForInterfaceSet (
    CONST UINTERFACESET *ifcset
        );
```

Description: Searches through the interfaces for the parent configuration until a match is found or the search is exhausted.

Returns: The index of the InterfaceSet if found, -1 if not found.

13.2.7 UsbFindNextDescriptorInConfig

Function: Generic routine for finding the next descriptor of a certain kind within a configuration descriptor.

Definition:

```
CONST USBIF_DESC_WIRE *UsbFindNextDescriptorInConfig (
        CONST USBIF_CFGDESC_WIRE *ConfigDesc,
        CONST VOID /*OPTIONAL */ *StartPosition,
        INT                      DescriptorCode
        );
```

Description: This routine is the engine that parses configuration descriptors in a robust way. ConfigDesc must point to the image of a configuration descriptor, including (followed by) the descriptor's associated descriptors. If StartPosition is NULL, the first matching descriptor after the configuration descriptor is returned.

Otherwise, StartPosition should be the result of a prior call to this routine; the search starts with the next descriptor.

The argument DescriptorType is used for matching. If it is -1, then the next descriptor after the current one will be matched; otherwise, descriptors with codes other than the specified DescriptorType will be skipped.

Returns: NULL if no match was found; otherwise pointer to the matching descriptor.

13.2.8 UsbFindNextDescriptorInInterface

Function: Generic routine for finding the next interface descriptor of a certain kind within a configuration descriptor.

Definition:

```
CONST USBIF_DESC_WIRE *UsbFindNextDescriptorInInterface(
        CONST USBIF_CFGDESC_WIRE *ConfigDesc,
        CONST VOID /*OPTIONAL */ *StartPosition,
```

```
    INT                     DescriptorCode
    );
```

Description: This routine is the engine that parses configuration descriptors in a robust way. DescriptorCode is used for matching, as described below:

ConfigDesc must point to the image of a configuration descriptor, including (followed by) the descriptor's associated descriptors. If StartPosition is NULL, the first matching descriptor after the configuration descriptor is returned.

Otherwise, if pStartInterface is non-NULL, it is taken to point to a descriptor within the configuration bundle given by pConfigDesc. For proper operation, this descriptor should either be an interface descriptor, or else it should be the non-NULL result of a prior call to this routine. The search starts with the next descriptor after pStartDesc, and will end (unsuccessfully) if an interface descriptor is encountered.

The argument DescriptorType is used to control the search. If it is -1, then the next descriptor after the current one will be matched; otherwise, it is taken as the 8-bit descriptor code for the descriptor of interest. Descriptors with codes other than the specified DescriptorType will be skipped.

The implementation is quite paranoid and is (supposed to be) guaranteed to terminate and not to touch invalid memory, provided the incoming pointers are mapped to memory that can be safely touched, and provided that the config descriptor's length field in fact represents how many bytes are present in the config bundle.

Returns: NULL if no match was found; otherwise pointer to the matching descriptor.

13.2.9 UsbPumpConfigBundle_FindNextClassDescInInterface

Function: Parse descriptors associated with an interface descriptor, locating class-specific descriptors.

Definition:

```
CONST USBIF_DESC_WIRE *UsbPumpConfigBundle_FindNextClassDescInInterface(
        IN CONST USB_CONFIGURATION_DESCRIPTOR *IfcDesc,
        IN CONST VOID *StartPosition,
        IN LONG DescriptorType,
        IN LONG DescriptorSubType
        );
```

Description: This routine starts with an interface descriptor. It searches among the subordinate descriptors for a descriptor with a matching descriptor type and subtype. As with ParseConfigurationDescriptor, a value of -1 supplied for either match code indicates that all values should be treated as a match.

If descriptor subtype is -1, then descriptors that are very short (only two bytes) are accepted; otherwise, it is required that any examined descriptor be at least 3 bytes long.

Function: Pointer to the extracted ClassDesc, or NULL.

Notes: Stops when an interface descriptor is encountered, which is right (according to the USB specification).

Despite the name, this can be used to search for any kind of descriptor that is attached to an interface (not only class-specific).

13.2.10 UsbParseConfigurationDescriptor

Function: Parse a configuration descriptor; enhanced, portable implementation of USBD_ParseConfigurationDescriptorEx.

Definition:

```
CONST USBIF_DESC_WIRE *UsbParseConfigurationDescriptor (
        CONST USBIF_CFGDESC_WIRE *ConfigDesc,
        CONST VOID /*OPTIONAL */ *StartPosition,
        INT                      InterfaceNumber,
        INT                      AlternateSetting,
        INT                      InterfaceClass,
        INT                      InterfaceSubClass,
        INT                      InterfaceProtocol
        );
```

Description: ConfigDesc must point to a configuration descriptor, followed by the descriptor's associated descriptors. If StartPosition is NULL, the first matching interface descriptor is returned. Otherwise, StartPosition should be the result of a prior call to this routine; the next matching interface descriptor is returned.

The arguments InterfaceNumber, AlternateSetting, InterfaceClass, InterfaceSubClass, and InterfaceProtocol are used for matching. If a match argument is -1, then any value for that field will be matched; otherwise, descriptors which do not match the specified setting will be skipped.

Returns: Pointer to the extracted IfcDesc, or NULL.

13.2.11 UsbFindNextDescriptorInBos

Function: Generic routine for finding the next descriptor of a certain kind within a BOS descriptor.

Definition:

```
CONST USBIF_DESC_WIRE * UsbFindNextDescriptorInBos(
        IN CONST USBIF_BOSDESC_WIRE *pBosDesc,
        IN CONST VOID *pStartPosition OPTIONAL,
        IN INT DescriptorCode
        );
```

Description: This routine is the engine that parses BOS descriptors in a robust way.

pBosDesc must point to the image of a BOS descriptor, including (followed by) the descriptor's associated descriptors. If pStartPosition is NULL, the first matching descriptor after the BOS descriptor is returned. Otherwise, pStartPosition should be the result of a prior call to this routine; the search starts with the next descriptor.

The argument DescriptorType is used for matching. If it is -1, then the next descriptor after the current one will be matched; otherwise, descriptors with codes other than the specified DescriptorType will be skipped.

The implementation is quite paranoid and is (supposed to be) guaranteed to terminate and not to page fault, provided that the incoming pointers are mapped to memory that can be legitimately touched, and provided that the BOS descriptor's length field in fact represents how many bytes are present.

Returns: NULL if no match was found; otherwise pointer to the matching descriptor.

13.3 Debugging Functions

13.3.1 UsbDebugLogf

Function: printf() version for logging; it is unconditional, and requires a platform pointer.

Definition:

```
VOID UsbDebugLogf (
    UPLATFORM *pPlatform,
    CONST TEXT *fmt,
    ...
    );
```

Description: This routine is used to unconditionally log a message to the platform logging stream. A message is formatted and passed to the platform.

Returns: Nothing.

Macro for invoking:

```
TTUSB_LOGF(args)
```

Notes: This function has been replaced by UsbPumpDebug_PlatformLogf(), and is provided only for backwards compatibility.

13.3.2 UsbDebugPrintf

Function: Conditional printf () -- logs a message if flag word masked with the USB debug dword is non-zero.

Definition:

```
VOID UsbDebugPrintf (
    UDEVICE *self,
    UINT32 mask,
    CONST TEXT *fmt,
    ...
    );
```

Description: If (self->udev_debugflags & mask) != 0, then the message controlled by *fmt and the following args is logged. Otherwise, no output is produced.

Returns: Nothing.

Macro for invoking:

```
TTUSB_PRINTF(args)
```

Notes: This function has been replaced by UsbPumpDebug_DevicePrintf(), and is provided only for backwards compatibility.

Table 11. Description of debug mask

Mask Name	Value	Description
UDMASK_FATAL_ERROR	0	pseudo level: fatal error
UDMASK_ERRORS	1L << 0	trace gross errors
UDMASK_ANY	1L << 1	catch-all category
UDMASK_FLOW	1L << 2	flow through the system

Mask Name	Value	Description
UDMASK_CHAP9	1L << 3	chapter 9 events
UDMASK_PROTO	1L << 4	protocol events
UDMASK_BUFQE	1L << 5	bufqe events
UDMASK_HWINT	1L << 6	trace hw interrupts
UDMASK_TXDATA	1L << 7	trace TX data
UDMASK_RXDATA	1L << 8	trace RX data
UDMASK_HWDIAG	1L << 9	trace HW diagnostics
UDMASK_HWEVENT	1L << 10	device event
UDMASK_VSP	1L << 11	vsp protocol
UDMASK_ENTRY	1L << 12	procedure entry/exit
UDMASK_ROOTHUB	1L << 13	root hub flow
UDMASK_USBDI	1L << 14	USBDI debug
UDMASK_HUB	1L << 15	hub class flow
UDMASK_DEVBASE_N	16	for building masks
UDMASK_HCD	1L << 16	hcd flow

13.3.3 UsbPumpDebug_PlatformLogf

Function: printf () version for logging; it is unconditional, and requires a platform pointer.

Definition:

```
VOID UsbPumpDebug_PlatformLogf(
UPLATFORM *pPlatform,
CONST TEXT *fmt,
);
```

Description: This routine is used to unconditionally log a message to the platform logging stream. A message is formatted and passed to the platform.

Returns: Nothing.

Notes: Prior to V1.79a, this function was called UsbDebugLogf (). The old name can be made available to older code by #defining __TMS_DATAPUMP_COMPAT_V1_6 to 1 on the compile command line.

13.3.4 UsbPumpDebug_DevicePrintf

Function: Conditional printf () -- logs a message if flag word masked with the USB debug dword is non-zero. Formerly called UsbDebugPrintf ().

Definition:

```
VOID UsbPumpDebug_DevicePrintf(
UDEVICE *pDevice,
UINT32 mask,
CONST TEXT *fmt,
...
);
```

Description: If the mask is zero, the message is unconditionally logged. Otherwise, if at least one of the bits in mask must also be set in pDevice->udev_ulDebugFlags, then the message is logged. Otherwise, no output is produced.

The format used is defined by UsbDebugVprintf ().

Returns: Nothing.

Notes: Prior to V1.79a, this function was called UsbDebugPrintf (). Use of mask == 0 to select unconditional printout was added in V1.79a.

See also: UsbDebugVprintf ().

13.3.5 UsbPumpDebug_ObjectPrintf

Function: Issue the IOCTL to an object that results in a debug print (assuming a checked build).

Definition:

```
VOID
UsbPumpDebug_ObjectPrintf(
   USBPUMP_OBJECT_HEADER *pObject,
   UINT32 Mask,
   CONST TEXT *pFmt,
   ...
   );
```

Description: Find the nearest print method, and UsbDebugVprintf () to display a message under the control of pFmt.

Returns: Nothing.

13.3.6 UsbPumpDebug_PlatformFlush

Function: Delay execution until all debug output has been pushed to the recipient.

Definition:

```
VOID UsbPumpDebug_PlatformFlush(
UPLATFORM *pPlatform
);
```

Description: Delays execution until all output has been transferred to the debug log device (if it makes sense to do so).

Returns: Nothing.

13.3.7 UsbDebugSnprintf

Function: Do UsbDebugPrintf () operation, sending output characters to sized buffer.

Definition:

```
int UsbDebugSnprintf(
TEXT *buf,
BYTES size,
CONST TEXT *fmt,
...
);
```

Description: UsbDebugSnprintf () performs a formatted output operation, writing the output characters into the buffer at (buf). Nbuf specifies the maximum output buffer size; the result is always terminated with a '\0', and space for this must be included. Therefore, strlen (buf) will always be in the range [0..nbuf-1]. Extra output (after the nbuf-1'th character) is counted for computing the result, but discarded.

Returns: (int) -1 to indicate errors (usually due to a bad format specification); otherwise the number of characters needed for the output. If the result is greater than or equal the size of the buffer, then the buffer was too small and some of the output was discarded.

If buf == NULL, the result is simply the size of the buffer that needs to be allocated to hold the result; caller must add 1for the trailing '\0', before using this to allocate an output buffer.

Bugs: For maximum portability (if code is also to use snprintf on other platforms), only compare the result to (int) -1 – other implementations vary on what they do if the buffer is too small.

A UsbDebugSprintf () is not provided, because it is considered not safe to use primitives that don't support buffer length checking.

13.3.8 UsbDebugVprintf

Function: Debug formatted print routine.

Definition:

```
INT UsbDebugVprintf (
    TTUSB_VPRINTF_FN *pfn,
    VOID             *functx,
    CONST TEXT       *fmt,
    va_list          av
    );
```

Description: Searches through the print operation, building up buffers full of data which are subsequently written to the output using calls of the form:

(*pfn)(funarg, buf, nc)

where buf is a pointer to a buffer, and nc is the number of bytes to write. (*pfn) must return nc if no errors occur, otherwise it writes the number of characters written. (Writing fewer characters than written is considered to be an error.) (BYTES) -1 is always taken as an error condition.

Formatting is done according to the rules given in Harbison & Steele, with some modifications in the direction of the C 1999 standard (ISO/IEC 9899:1999).

Returns: Number of characters actually written, or -1 if an error.

13.3.9 UsbPumpDeviceLinkState_Name

Function: Diagnostic API function; returns the name corresponding to a given USB DataPump Device Link State code..

Definition:

```
#include "udevice.h"

      CONST TEXT *
            UsbPumpDeviceLinkState_Name(
            UDEVICE_LINK_STATE  LinkStateCode,
            );
```

Description: This routine is not used by DataPump code (except in debug mode), but is provided as part of the common library for use by client functions. It returns a

pointer to the text equivalent of LinkStateCode, or to a constant string "<<unknown-link-state>>".

Returns: Non-NULL pointer to an immutable name string.

13.4 Mapping Functions

13.4.1 UsbFindEndpointByAddr

Function: Given an endpoint address, search the hardware endpoint array and find the matching UENDPOINT.

Definition:

```
UENDPOINT *UsbFindEndpointByAddr (
    CONST UDEVICE *udev,
    unsigned epaddr
    );
```

Description: This function looks through the endpoint array, searching for a match. The search is linear. The endpoint array is assumed to contain all the addressable endpoints; each endpoint must point back to a pipe, including the (two) endpoints that correspond to the default pipe.

Holes in the endpoint sequence are allowed, as long as the pipe pointer is NULL.

Returns: Nothing.

13.4.2 UsbFindInterfaceByAddr

Function: Locate the UINTERFACE* for a specific interface of a device.

Definition:

```
UINTERFACE *UsbFindInterfaceByAddr (
        CONST UDEVICE *self,
        unsigned ifcAddr
        );
```

Description: ifcAddr is a value as passed in a wIndex field when trying to select a specific interface using a SETUP packet. In the context of the currently selected configuration, this function finds the interface structure that matches, or returns NULL if no matches are found.

Returns: Pointer to interface; or NULL if an error has occurred.

13.5 DCD API

The functions in this section are the primary API by which the DCDs indicate significant state changes to the DataPump core. These functions are mostly used in writing DCD and must not be used by code in other layers.

13.5.1 UsbProcessAttach

Function: Notify DataPump of hardware-level Vbus attach.

Definition:

```
VOID UsbProcessAttach(
    UDEVICE *self
    );
```

Description: The ATTACH event is broadcast to the hardware notification queue by calling UDEVREPORTEVENT and to the portable device notification queue by calling UsbReportEvent, respectively.

This routine is a primary export from the DataPump to the DCDs. The DCD is expected to call this procedure after detecting a USB bus attach. The DCD is required not to call this routine spuriously.

DCDs must not call UsbReportDeviceEvent directly; instead they must call UsbProcessAttach()to allow the DataPump to perform additional filtering.

Returns: No explicit result.

13.5.2 UsbProcessDetach

Function: Broadcast USB bus detach from DCD inward.

Definition:

```
VOID UsbProcessDetach(
    UDEVICE *self
    );
```

Description: This function tears down the current configuration, and then broadcasts the event at the device level.

This routine is a primary export from the DataPump to the DCD. The DCD is expected to call this procedure after detecting a USB bus reset.

Returns: No explicit result.

13.5.3 UsbProcessControlPacket

Function: Process "unknown" control packets -- i.e., any packets not already processed at the hardware layer.

Definition:

```
BOOL UsbProcessControlPacket (
    UDEVICE *self,
    UINT8 *setup
    );
```

Description: The setup packet is parsed, and dispatched to the appropriate event queue using an UEVENTSETUP structure to carry the data.

The setup packet is unpacked into a USETUP structure, and passed to the appropriate layer based on the values of bmRequestType and bRequest.

The event processing routines are required to set uec_accept TRUE to accept; and to set uec_reject TRUE to reject. Both start out FALSE; and the event processing routines should set the accept field to accept, set the reject field to reject; or else leave both fields alone, so the accept field becomes the logical OR of all the accepts, and the reject field becomes the logical OR of all the rejects.

Returns: TRUE if the packet is OK; in that case, the client has replied. FALSE if the packet is to be STALLED; it is the caller's job to return the STALL.

13.5.4 UsbProcessGetConfiguration

Function: Request DataPump to process inbound GET_CONFIGURATION request.

Definition:

```
VOID UsbProcessGetConfiguration(
    UDEVICE *self,
    CONST USETUP *pSetup
    );
```

Description: The current configuration is computed and returned to the host via suitable callbacks to the DCD.

Returns: Nothing.

13.5.5 UsbProcessGetDescriptor

Function: Request DataPump to process inbound GET_DESCRIPTOR request.

Definition:

```
VOID UsbProcessGetDescriptor(
    UDEVICE *self,
    CONST USETUP *pSetup
    );
```

Description: The specified descriptor is read from the tables.

A complicated method is used to support externally stored descriptors:

1. The function searches to find the descriptor value in the ROM (static) descriptor tables.

2. If this is a string descriptor read, and the user supplied an "external descriptor" bit table, and the descriptor is NOT marked external, the descriptor data is passed back to the host (from ROM).

3. Otherwise, if the user supplied a "get descriptor" filter function, it is called as follows:

```
(*pfn)(self,
        bType,
        bIndex,
        wIndex,  -- langid or zero
        index-this-call,
        pointer-to-ROM-descriptor,
        length-of-ROM-descriptor,
        pointer-to-RAM-buffer,
        bytes-in-RAM-buffer,
        pointer-to-result-pointer OUT,
        pointer-to-result-length OUT
        );
```

The routine is expected to set *pointer-to-result-pointer to point to the data returned, and set *pointer-to-result-length to the number of bytes to be transmitted. It will return:

- 0 (USBPUMP_GETDESCRFILTER_PASS) if the core pump is to return the ROM descriptor.

- 1 (USBPUMP_GETDESCRFILTER_SEND) if the core pump is to transmit the result from *pointer-to-result-pointer, *pointer-to-result-length. This can also be used to forcibly stall the request, by setting *pointer-to-result-pointer to NULL. Since it is initialized to null before calling the function, returning without setting the pointer will always cause a stall.

- 2 (USBPUMP_GETDESCRFILTER_FRAG) if the core pump is to transmit the result, and then ask for more data (with index-this-call suitably advanced by the number of bytes previously transmitted) [NOT YET SUPPORTED]

- 3 (USBPUMP_GETDESCRFILTER_FAIL) if the core DataPump is to return an error (STALL endpoint)

- If the routine doesn't want to filter this descriptor, it can just return 0, and the DataPump will take the appropriate action.

Note: index-this-call is an argument provided for future use; it is intended to allow the same user-function to be used in a future version of the DataPump that supports constructing the reply packet-by-packet, instead of in a single buffer. This is not implemented yet.

Returns: TRUE if this routine handled the descriptor;
FALSE if this routine determined that it could not handle the descriptor. In these cases, the caller is responsible for replying.

13.5.6 UsbProcessGetDeviceStatus

Function: Request the DataPump to process an incoming GET_STATUS request.

Definition:

```
VOID UsbProcessGetDeviceStatus(
    UDEVICE *self,
    CONST USETUP *pSetup
    );
```

Description: Decode a device status request, and build a reply in the device's reply buffer. Send it back, if possible. To get the information, the DataPump performs a down-call back to the DCD layer using the device-event mechanism.

Returns: Nothing.

13.5.7 UsbProcessGetEndpointStatus

Function: Request the DataPump to process an incoming GET STATUS (ENDPOINT) command.

Definition:

```
VOID UsbProcessGetEndpointStatus(
    UDEVICE *self,
```

```
    CONST USETUP *pSetup
    );
```

Description: The setup is validated, and the endpoint is located and then decoded. The status is returned, or else a stall is returned to the host.

Returns: Nothing.

13.5.8 UsbProcessGetInterface

Function: Request the DataPump to process an incoming GET INTERFACE command.

Definition:

```
VOID UsbProcessGetInterface (
    UDEVICE *self,
    CONST USETUP *pSetup
    );
```

Description: The interface setting is calculated and returned to the host, if possible.

Returns: Nothing.

13.5.9 UsbProcessGetInterfaceStatus

Function: Request the DataPump to process an incoming GET_STATUS(interface).

Definition:

```
VOID UsbProcessGetInterfaceStatus(
    UDEVICE *self,
    CONST USETUP *pSetup
    );
```

Description: The interface is located, and if valid, a report is built that is zero. It is broadcast by calling UsbReportInterfaceEvent () to the interface, filled in, then send back to the host.

Returns: Nothing.

13.5.10 UsbProcessResume

Function: Notify DataPump of USB resume.

Definition:

```
VOID UsbProcessResume(
    UDEVICE *self
    );
```

Description: The RESUME event is broadcast to the hardware notification queue, and to the portable device notification queue.

This routine is a primary export from the USB DataPump to the hardware interface layer. The interface layer is expected to call this procedure after detecting a USB bus resume.

Note, however, the interface layer must have already put the system into run mode.

Returns: Nothing.

13.5.11 UsbProcessSetAddress

Function: Request the DataPump to process a set-address SETUP packet.

Definition:

```
BOOL UsbProcessSetAddress (
    UDEVICE *self,
    CONST USETUP *pSetup,
    UINT8 *setup
    );
```

Description: SetAddress causes cancellation of all pending I/Os for endpoints and unconditionally moves to the unconfigured state.

When an address change arrives from the host, it needs to be acted upon to force the device into the unconfigured state. This routine does that, sending messages to all relevant recipients.

The configuration-change messaging policy is set by UsbChangeConfig(). After the messages for change-config have been sent, a "set address" is reported to the hardware by calling UDEVREPORTEVENT and to the global device notification queue, which is pDevice->udev_noteq.

To actually change the address, a message is sent to the device report switch (UEVENT_SETADDR_EXEC).

This routine is a primary export from the USB DataPump to the DCD. The interface layer is expected to call this procedure after receiving and decoding a setup message, but before accepting the packet (by allowing the final handshake to complete). If the message is refused, the HIL must return a STALL response to

the host (in order to be compliant). The HIL is responsible for actually changing the address.

Returns: TRUE, to indicate that the packet is OK. This means that the HIL should proceed to the handshake stage and complete the address acceptance.

13.5.12 UsbProcessSetClearDevFeature

Function: This routine is called from the DCD, to process a Set/Clear device feature packet.

Definition:

```
BOOL UsbProcessSetClearDevFeature (
    UDEVICE *self,
    ARG_USHORT feature,
    ARG_BOOL value,
    UINT8   *setup
    );
```

Description: Remote wakeup state is recorded, and the event is propagated.

This routine is a primary export from the USB DataPump to the DCD. The interface layer is expected to call this procedure after receiving and decoding a setup message, but before accepting the packet (by allowing the final handshake to complete). If DataPump refuses the message, the HIL must return a STALL response to the host (in order to be compliant).

Returns: TRUE if the feature command was correct and was processed; FALSE otherwise. If a FALSE is returned, then the command should be stalled.

13.5.13 UsbProcessSetClearEpFeature

Function: Handle "set/clear endpoint feature".

Definition:

```
BOOL UsbProcessSetClearEpFeature (
    UDEVICE *self,
    ARG_USHORT ep,
    ARG_USHORT feature,
    ARG_BOOL value,
    UINT8 *setup
    );
```

Description: This routine filters and handles the "Set/Clear endpoint feature" passing it off to both the hardware filter and the application event queue.

This routine is a primary export from the USB DataPump to the DCD. The interface layer is expected to call this procedure after receiving and decoding a setup message, but before accepting the packet (by allowing the final handshake to complete). If the message is refused, the HIL must return a STALL response to the host (in order to be compliant).

Returns: FALSE if a STALL result is to be returned; TRUE otherwise.

13.5.14 UsbProcessSetClearIfcFeature

Function: Handle "set/clear interface feature".

Definition:

```
BOOL UsbProcessSetClearIfcFeature (
    UDEVICE *self,
    ARG_USHORT ifcnum,
    ARG_USHORT feature,
    ARG_BOOL value,
    UINT8 *setup
    );
```

Description: This routine filters and handles the "Set/Clear interface feature" passing it off to both the hardware filter and the application event queue.

This routine is a primary export from the USB DataPump to the DCD. The interface layer is expected to call this procedure after receiving and decoding a setup message, but before accepting the packet (by allowing the final handshake to complete). If the message is refused, the HIL must return a STALL response to the host (in order to be compliant).

Returns: FALSE if a STALL result is to be returned; TRUE otherwise.

13.5.15 UsbProcessSetConfig

Function: This routine implements the USB DataPump's response to a set config message.

Definition:

```
BOOL UsbProcessSetConfig (
    UDEVICE *self,
    ARG_USHORT config,
    UINT8 *setup
    );
```

Description: When a configuration change arrives from the host, the user needs to check the configuration, and then adjust the data structures to reflect the change in

configuration if the configuration should be changed. This routine does that, sending messages to all relevant recipients.

The messaging policy is set by UsbChangeConfig.

This routine is a primary export from the USB DataPump to the DCD. The interface layer is expected to call this procedure after receiving and decoding a setup message, but before accepting the packet (by allowing the final handshake to complete). If the configuration is refused, the HIL must return a STALL response to the host (in order to be compliant).

Returns: TRUE if the configuration was accepted, FALSE otherwise.

13.5.16 UsbProcessSetDescriptor

Function: Handle set descriptor for the portable DataPump.

Definition:

```
VOID UsbProcessSetDescriptor (
    UDEVICE *self,
    CONST USETUP *pSetup
    );
```

Description: For a request to set descriptor, first, the specified descriptor is first selected from the tables. If the specified descriptor is allowed to change, the process is started.

Since the setup packet only has part of the information, more data is required from the host. The user must supply a "set descriptor" filter function, otherwise, I/O is stalled.

The filter function is called as follows:

```
(*pfn)(self,
    bType,
    bIndex,
    wIndex,  -- langid or zero
    wLength, -- overall length
    index-this-call,
    pointer-to-ROM-descriptor,
    length-of-ROM-desriptor,
    pointer-to-buffer HOST,
    pointer-to-buffer-length
    );
```

This routine is expected to go through the host packet and identify next steps. It will return:

- 0 (USBPUMP_SETDESCRFILTER_GETNEXT) if more data is expected from the host, the user places a buffer in queue to wait for data;
- 1 (USBPUMP_SETDESCRFILTER_DONE) if sufficient data is received;
- 2 (USBPUMP_SETDESCRFILTER_ERROR) if an error is detected.

Returns: Nothing.

13.5.17 UsbProcessSetInterface

Function: Handles the USB "Set Interface" message.

Definition:

```
BOOL UsbProcessSetInterface(
    UDEVICE *self,
    ARG_USHORT interface,
    ARG_USHORT index,
    UINT8 *setup
    );
```

Description: When an interface change arrives from the host, the message needs to be checked, and the data structures adjusted to reflect the change in interface settings if the settings should be changed. This routine does that, sending messages to all relevant recipients.

The messaging policy is set by UsbChangeInterface).

This routine is a primary export from the USB DataPump to the DCD. The HIL is expected to call this procedure after receiving and decoding a setup message, but before accepting the packet (by allowing the final handshake to complete). If the message is refused, the HIL must return a STALL response to the host (in order to be compliant).

Returns: TRUE if message was accepted, FALSE if it was rejected.

13.5.18 UsbProcessSetupPacketRaw

Function: Process setup packet for DCD.

Definition:

```
VOID UsbProcessSetupPacketRaw (
    UDEVICE *self,
    UINT8 *pkt
    );
```

Description: This routine processes a single 8-byte setup packet, found at *pkt. The setup packet is decoded according to the rules given in Chapter 9 of the USB Core Specification; this routine then calls the appropriate UsbProcess... routine. It uses the UDEVICE's UDEVREPLY () command to indicate that processing of the setup packet is complete -- this might occur asynchronously to this call.

Returns: No explicit result.

Porting Notes:

This routine is optimized to work with device controllers such as the FT900 USB device controller, that have no intelligence of their own. DCDs for smarter hardware can be implemented in two ways: either use this routine for decoding commands that aren't supported in hardware, or else call other UsbProcess... routines directly as needed. The former technique is easy, but may result in extra unused code being incorporated into the end product.

DCDs must tell the DataPump about SET_CONFIG, SET_INTERFACE and other Chapter 9 state-change events. One way to do this is to reconstruct the setup packet based on hardware state changes; another method is to call the appropriate UsbProcess... routine.

13.5.19 UsbProcessSuspend

Function: Notify DataPump of USB suspend.

Definition:

```
VOID UsbProcessSuspend (
    UDEVICE *self
    );
```

Description: The SUSPEND event is broadcast through report event function to the hardware notification queue, and to the portable device notification queue.

This routine is a primary export from the USB DataPump to the DCD. The DCD is expected to call this procedure after detecting a USB bus suspend.

Returns: Nothing.

13.5.20 UsbProcessUsbReset

Function: Broadcast USB bus reset function from hardware level inward.

Definition:

```
VOID UsbProcessUsbReset (
    UDEVICE *self
    );
```

Description: This function tears down the current configuration, and broadcasts the event at the device level.

This routine is a primary export from the USB DataPump to the DCD. The interface layer is expected to call this procedure after detecting a USB bus reset.

Returns: Nothing.

13.5.21 UsbProcessUsbResetV2

Function: Broadcast USB bus reset function from hardware level inward.

Definition:

```
VOID UsbProcessUsbResetV2(
    UDEVICE *pDevice,
    USBPUMP_DEVICE_SPEED newSpeed
    );
```

Description: This routine is called from high-speed DCDs when a USB reset event occurs.

This function tears down the current configuration, and then broadcasts the event at the device level.

This routine is a primary export from the USB DataPump to the DCD. The interface layer is expected to call this procedure after detecting a USB bus reset.

Returns: Nothing.

13.6 DCD Support Functions

13.6.1 UsbEpswEpEventStub

Function: A stub endpoint event function to be used if an event is ever delivered to an unprepared endpoint.

Definition:

```
VOID UsbEpswEpEventStub (
    UDEVICE *self,
    UENDPOINT *ep
    );
```

Description: This function simply returns. It exists to prevent useless calls to NULL. In debug mode, an error message is printed.

Returns: No explicit result.

13.7 Timer API

When polling or otherwise operating finite state machines, there is a need for periodic events driven from a reference source. This timing may be necessary, even if a given HCD is not operating. Rather than depend on HCD capabilities timing is provided, based on the platform's timer.

The basic paradigm is similar to using an asynchronous IOCTL to send a timer request to the `UPLATORM`. The platform simply delays the completion until the specified time has elapsed. Timers are always dispatched from the event loop. The timer objects support the following operations:

- Initialize

```
VOID UsbPumpTimer_Initialize (
   UPLATFORM *pPlatform,
   USBPUMP_TIMER *pTimerObject,
   USBPUMP_TIMER_DONE_FN *pDoneFn
   );
```

- To start a timer:

```
USTAT UsbPumpTimer_Start (
   UPLATFORM *pPlatform,
   USBPUMP_TIMER *pTimerObject,
   ARG_USBPUMP_TIMER_TIMEOUT nMillisecs,
   USBPUMP_MILLISECONDS *pStartTime OPTIONAL
   );
```

  The result returned is the error code, which will be either `USTAT_OK` or an appropriate error code.

  If `pStartTime` is not `NULL`, `*pStartTime` is set to the value of the system clock at the time the timer was started, in milliseconds. i.e., NULL means "now".

  When the timer has completed, the specified `pDoneFn` is called using the following prototype:

```
VOID (*pDoneFn)(
   UPLATFORM *pPlatform,
   USBPUMP_TIMER *pTimerObject,
   USBPUMP_MILLISECONDS CurrentTickCounter
   );
```

  The current tick counter is the time just before `pDoneFn` is entered, which allows relatively easy implementation of periodic timers without having to call back to the system to find the time.

- Cancel – of course, this operation always suffers from a race condition.

```
BOOL UsbPumpTimer_Cancel (
   UPLATFORM *pPlatform,
   USBPUMP_TIMER *pTimerObject
   );
```

The specified timer is cancelled. The result is TRUE if the timer was cancelled (and will not fire), FALSE if the timer has already completed. (The way to think about this is that the postcondition, if the result is TRUE, is that the completion routine has not yet been called and will not be scheduled.)

Timers contain the following fields:

Table 12. USBPUMP_TIMER Contents

Field	Description
`USBPUMP_TIMER_DONE_FN *pDoneFn`	Completion function: this is the same as `VOID (*pDoneFn)(USBPUMP_TIMER *);`
`USBPUMP_TIMER *pNext, *pLast`	queue links
`USHORT QueueIndex`	index to queue head (internal private, used for cancellation)
`USHORT Ticks`	Ticks remaining, in queue increments (this cannot be examined by client software while the timer is running, and might not be in milliseconds – it is an implementation decision)

Low-level HCDs use a central service for timing out USBPUMP_HCD_REQUESTs. The common HCD platform object contains a timer object, and a sorted list of objects to time-out. For efficiency, the HCD maintains three queues for initial timeout <= 100ms, <= 1000ms, and >= 1000 ms timeouts. Objects are always inserted into the tail of the queue that is appropriate, and so will timeout after the specified period of time. The 10-100ms queue gets run every 10ms; the 100-1000ms gets run every 50ms, the 1000 and up queue gets run every 500ms. Objects never get moved from their queue, so the resolution of the timeout does go down as timeouts go up.

Cancellation is achieved by removing the object from the queue it is in. If QueueIndex is 0, then the timer is not in a queue. Synchronization is not a problem, because the platform is constrained to update the timeout queue from inside the DataPump context. The caller is required to embed the timer block in a larger structure in order to obtain any back context.

QueueIndex and QueueTicks are defined for use by the default timer implementation. If the platform layer substitutes a different timer mechanism, then the platform layer might reuse these fields for its own purposes.

13.7.1 Timer Implementation Framework

Timer support is added to the platform by adding two fields to the UPLATFORM, described in Table 13.

Table 13. UPLATFORM additions for timer support

Field	Description
`CONST USBPUMP_TIMER_SWITCH *pPlatform->upf_pTimerSwitch;`	Pointer to table of dispatch functions.
`VOID *pPlatform->upf_pTimerContext;`	Context pointer for timer implementation.

The USBPUMP_TIMER_SWITCH contents are described in Table 14.

Table 14. USBPUMP_TIMER_SWITCH Contents

Field	Description
`USBPUMP_TIMER_SYSTEM_INITIALIZE_FN *pTimerSystemInitialize;`	Function for initializing the timer system.
`USBPUMP_TIMER_UPCALL_TICK_FN *pTimerUpcallTick;`	Notification function for timer updates.
`USBPUMP_TIMER_INITIALIZE_FN *pTimerInitialize;`	Method for `UsbPumpTimer_Initialize`
`USBPUMP_TIMER_START_FN *pTimerStart;`	Method for `UsbPumpTimer_Start`
`USBPUMP_TIMER_CANCEL_FN *pTimerCancel;`	Method for `UsbPumpTimer_Cancel`
`USBPUMP_TIMER_TICK_START_FN *pTimerTickStart;`	Function for start timer tick interrupt.
`USBPUMP_TIMER_TICK_STOP_FN *pTimerTickStop;`	Function for stop timer tick interrupt.

The types are described in subsequent sections.

The timer context pointer is provided for two reasons:

1. It allows for replacement of the DataPump standard timer implementation with a platform specific one, in case the DataPump's implementation is not suitable.

2. It allows for omission of timer support when it is not required. For example, device-only applications of the DataPump typically do not need timer support.

13.7.1.1 USBPUMP_TIMER_INITIALIZE_FN

This function is the implementation for `UsbPumpTimer_Initialize`. The parameters and behavior are the same.

```
typedef VOID USBPUMP_TIMER_INITIALIZE_FN(
          UPLATFORM *pPlatform,
          USBPUMP_TIMER *pTimerObject,
          USBPUMP_TIMER_DONE_FN *pDoneFn
          );
```

13.7.1.2 USTAT USBPUMP_TIMER_START_FN

This function is the implementation for `UsbPumpTimer_Start`. The parameters and behavior are the same.

```
USTAT USBPUMP_TIMER_START_FN(
          UPLATFORM *pPlatform,
          USBPUMP_TIMER *pTimerObject,
          ARG_USHORT nMillisecs,
          USBPUMP_MILLISECONDS *pStartTime OPTIONAL
          );
```

13.7.1.3 USBPUMP_TIMER_CANCEL_FN

This function is the implementation for `UsbPumpTimer_Cancel`. The parameters and behavior are the same.

```
BOOL USBPUMP_TIMER_CANCEL_FN(
          UPLATFORM *pPlatform,
          USBPUMP_TIMER *pTimerObject
          );
```

13.7.1.4 USBPUMP_TIMER_UPCALL_TICK_FN

This function has the following prototype:

```
typedef VOID
USBPUMP_TIMER_UPCALL_TICK_FN(
          UPLATFORM *pPlatform,
          USBPUMP_MILLISECONDS CurrentTime
          );
```

This entry is provided for use by the platform layer. It should be called periodically from within DataPump context, to indicate that time has passed.

As with the DataPump event queues, a standard implementation of timers is shipped as part of the DataPump library. This implementation assumes periodic updates (via `USBPUMP_TIMER_UPCALL_TICK_FN`) from the system, normally via a timer interrupt of some kind. Resolution is not critical.

The platform layer might need a different implementation of the timer system. In this case, the platform layer need not supply or use this function. (For example, the platform layer might directly translate timers into native operating-system equivalent operations.)

13.8 Miscellaneous Functions

All of the following functions are defined in the USB DataPump header file, "usbpump.h".

13.8.1 UsbCopyAndReply

Function: Utility for SETUP processing routines -- copy a result to a safe place, and then use it to reply.

Definition:

```
USHORT UsbCopyAndReply (
    UDEVICE *pSelf,
    VOID *pDeviceBuffer,
    USHORT size_DeviceBuffer,
    CONST UCHAR *pAnyBuffer,
    USHORT length_AnyBuffer,
    USHORT wLength
    );
```

Description: This routine is useful for those procedures that prepare replies, but are not sure if the buffer is addressable. The data is copied to a supplied bounce-buffer, and then sent on to the host.

Returns: Actual number of bytes transferred.

13.8.2 UsbDeviceReply

Function: Portable routine, which centralizes common bookkeeping for USB control endpoint replies using UDEVICE_REPLY ().

Definition:

```
USHORT UsbDeviceReply (
    UDEVICE *pSelf,
```

```
    CONST VOID *pReply,
    USHORT ReplyMax,
    USHORT ReplyActual
    );
```

Description: This routine prepares a device reply using UDEVICE_REPLY, taking into account max vs. actual.

Returns: The reply length actually used.

Notes: ReplyMax should be set to the original wLength of the setup packet.

13.8.3 UsbPumpLib_BufferCompareString

Function: Compare buffer to string for equality.

Definition:

```
BOOL
UsbPumpLib_BufferCompareString(
   CONST TEXT *buf,
   BYTES n,
   CONST TEXT *s
   );
```

Description: In order for the inputs to be considered equal, each character in buf must match the corresponding character in s; and UHIL_lenstr(s) must be equal to n.

Returns: TRUE if match, FALSE otherwise.

Notes: Returns TRUE if the pointers are equal, otherwise returns FALSE if either pointer is NULL.

13.8.4 UsbPumpLib_BufferFieldIndex

Function: Locate a field in a buffer.

Definition:

```
BYTES
UsbPumpLib_BufferFieldIndex(
   CONST TEXT *buf,
   BYTES n,
   BYTES i,
   BYTES fieldnum,
   int fsep
   );
```

Description: UsbPumpLib_BufferFieldIndex computes the index of the (fieldnum)'th field in an (n)-character buffer, starting at position (i).

If (fsep) is positive, then the buffer is considered to be composed of a number of fields separated by a magic character, fsep. We scan forward to the first character after the (fieldnum)th separator, or to the end of the buffer. Null fields are possible in this case; the zero'th field begins at index (i).

If (fsep) is negative, then the buffer is considered to be composed of a number of fields separated by whitespace. Field zero starts at the first non-white character after i; field one starts at the first non-white character after the first white-space character after field zero; and so forth. Null fields are impossible in this case, except at the end of a buffer.

Returns: The index of the specified field; this will be in [0,n-1] if the field exists, or [n] if the field wasn't found.

(n) is returned in case an error occurs, e.g. (i > n) or (buf == NULL).

Notes: Whitespace is blank, \t, \f, \n or \v.

Bugs: The characters in the buffer are treated as unsigned when comparing to fsep.

Whitespace is identical to being a character with value <= 0x20, or being DEL (0x7F).

13.8.5 UsbPumpLib_BufferFieldLength

Function: Find length of field at specified position in buffer.

Definition:

```
BYTES
UsbPumpLib_BufferFieldLength(
   CONST TEXT *buf,
   BYTES n,
   BYTES i,
   INT fsep
   );
```

Description: UsbPumpLib_BufferFieldLength() complements UsbPumpLib_BufferFieldIndex() (q.v.); it returns the length of the field, starting at position (i), with the delimiter specified by (fsep).

If (fsep) is positive, the buffer is considered to consist of a sequence of fields delimited by a magic character. A field is scanned forward from the specified position until the next delimiter is found, or until the end of the buffer is reached. Empty fields are possible in this case.

If (fsep) is negative, the buffer consists of a sequence of fields separated by arbitrary non-empty sequences of whitespace. Empty fields are not possible except at the end of the buffer. In this case, any leading whitespace is skipped,

then by skipping to the end of the next field; the number of characters skipped is returned.

Returns: UsbPumpLib_BufferFieldLength() returns the length of the field which starts at position (i). If any errors are detected (i.e., buf == NULL or i >= n), UsbPumpLib_BufferFieldLength() returns 0.

Notes: Whitespace is blank, \t, \f, \n or \v.

13.8.6 UsbPumpLib_CalculateMaxPacketSize

Function: Calculate the effective max packet size given info from an endpoint descriptor.

Definition:

```
INT32
UsbPumpLib_CalculateMaxPacketSize(
   USBPUMP_OBJECT_HEADER *pObjHdr,
   USBPUMP_DEVICE_SPEED devSpeed,
   UINT8 bmAttributes,
   ARG_UINT16 wMaxPacketSize,
   BOOL fStrict
   );
```

Description: Based on the device speed and the endpoint type, wMaxPacketSize is converted to a "safe-and-sane" max packet size. Note that the "extra packet" bits are left in-place if this is a periodic high-speed endpoint, or zeroed otherwise.

If fStrict, UsbPumpLib_CalculateMaxPacketSize() will return -1 for anything suspicious; otherwise it will try to substitute reasonable values where possible.

Returns: Either the max packet size, or -1.

13.8.7 UsbPumpLib_MatchPattern

Function: Match with simple wildcarding.

Definition:

```
BOOL
UsbPumpLib_MatchPattern(
   CONST TEXT *pPattern,
   BYTES nPattern,
   CONST TEXT *pValue,
   BYTES nValue
   );
```

Description: Compare the literal string buffer given by (pValue, nValue) to the pattern given by (pPattern, nPattern).

The comparison is a simple byte-by-byte comparison, except that the pattern character '*' is special; it matches zero or more arbitrary bytes. The comparison is anchored (in the sense of a general regular expression match) at the beginning and end; of course, an un-anchored match can be formed by beginning and ending the pattern string with '*'. So, for example, a pattern of 'a*b' matches 'ab', 'a...b', etc; but doesn't match 'a*bc'. However, a pattern of '*a*bc*' will match '...a....bc...', 'abc', 'a..bc..', etc.

As it does simple byte-by-byte comparision, it's CASE-SENSITIVE; hexadecimal 'a' doesn't match to hexadecimal 'A' especially when trying to match hexadecimal number literal.

Returns: TRUE for success, FALSE for failure.

Bugs: This routine is not suitable for use with international multi-byte-character input strings.

This routine cannot match a literal '*' in the value string.

13.8.8 UsbPumpLib_SafeCopyBuffer

Function: A Memory copy routine that is reasonably safe to use, i.e., to avoid buffer overflow during the copy procedure.

Definition:

```
BYTES
UsbPumpLib_SafeCopyBuffer(
   PVOID pOut,          /* base of output buffer */
   BYTES OutSize,       /* size of output buffer */
   BYTES OutIndex,      /* where to start copying within buffer */
   CONST VOID *pIn,     /* input buffer */
   BYTES InSize         /* size of data to copy */
   );
```

Description: This routine copies memory from the input buffer to the to the output buffer, taking into account the allocation size of the output buffer.

pOut is a buffer that has been allocated OutSize bytes. pIn points to a buffer with InSize bytes of information. OutIndex specifies the start position in the buffer.

Up to InSize bytes will be copied from pIn to pOut+OutIndex; but in no case will data be written outside the range of bytes pOut[0..OutSize)[5]. The copy size is reduced, to zero if necessary, to enforce this constraint.

If pIn is NULL, the specified portion of the output buffer is set to 0.

[5] [0, n) denotes the range from 0 to n-1

If pOut is NULL, no copy or fill is performed; however, the result is the number of bytes that would have been copied. This option is useful for determining what SafeCopyBuffer would have done if the pointer had not been NULL; it also ensures that loops using the result of SafeCopy are more likely to terminate even if handed a null output pointer.

Returns: Number of bytes actually copied (or that would have been copied except that pOut was NULL).

13.8.9 UsbPumpLib_SafeCopyString

Function: String copy routine that is reasonably safe to use.

Definition:

```
BYTES
UsbPumpLib_SafeCopyString(
   TEXT *pBuffer,
   BYTES nBuffer,
   BYTES iBuffer,
   CONST TEXT *pString
   );
```

Description: This routine copies memory from the input string to the given offset in the buffer, and appends a '\0', taking into account the size of the buffer.

pBuffer is a buffer that has nBuffer bytes allocated to it.

pString points to a nul-terminated string (ANSI, UTF-8, etc --encoding is not critical as long as '\0' always designates the end of the string.

Bytes from pString are copied to pBuffer+iBuffer. In no case will data be written outside the range of bytes pBuffer[0..nBuffer).

The resulting string at pBuffer+iBuffer is guaranteed to be NULL-terminated. Therefore, the maximum string size that can be handled without truncation is (nBuffer - iBuffer - 1) bytes long.

Boundary conditions can be considered without loss of generality by considering only the case where iBuffer == 0.

If pBuffer == NULL, pString == NULL or nBuffer == 0, then the result is always 0.

if nBuffer == 1, then the result is also always 0, but pBuffer[0] is set to '\0'.

If nBuffer > strlen(pString), then the entire string is copied to pBuffer, and a trailing '\0' is provided.

If nBuffer == strlen(pString), then all but the last byte is copied, a trailing '\0' is provided, and the result is (nBuffer - 1), or equivalently strlen(pString)-1.

Returns: Number of bytes of pString placed into the buffer.

The result + iBuffer is always less than nBuffer (in order to guarantee a trailing '\0'), unless nBuffer is zero.

Notes: If (iBuffer + the result) >= nBuffer, then it should be assumed that one or more bytes of the string were truncated. If nBuffer>0, and iBuffer + the result == nBuffer-1, then the string may have been truncated.

13.8.10 UsbPumpLib_ScanBuffer

Function: Scan a buffer looking for a matching byte.

Definition:

```
BYTES UsbPumpLib_ScanBuffer(
   const TEXT *b,
   BYTES n,
   TEXT c
   );
```

Description: UsbPumpLib_ScanBuffer() searches a buffer for the first occurrence of the specified byte, returning the byte index if found. If not found, the length of the buffer is returned.

Returns: Index of match, or length of buffer.

13.8.11 UsbPumpLib_ScanString

Function: Scan a string looking for a matching byte.

Definition:

```
BYTES UsbPumpLib_ScanString(
   CONST TEXT *s,
   TEXT    c
   );
```

Description: UsbPumpLib_ScanString () searches a string for the first occurrence of the specified byte, returning the byte index if found. If not found, the length of the string is returned.

Returns: Index of match, or length of buffer.

13.8.12 UsbPumpLib_UlongToBuffer

Function: Convert unsigned long to text in buffer.

Definition:

```
BYTES
UsbPumpLib_UlongToBuffer(
   TEXT *buf,
   BYTES n,
   ULONG ulnum,
   int radix
   );
```

Description: The number ulnum is converted to a string of ASCII characters in the buffer at buf, in the base specified by radix.

If radix is positive, then ulnum is interpreted as an unsigned long in the specified base. Radix must be in [2,16]; otherwise it is interpreted as base 10.

If radix is negative, then ulnum is interpreted as a signed long. If ulnum (as a LONG) is positive, it is output with a leading '+'; if ulnum is (as a LONG) is negative, it is output with a leading '-'. A sign will always be at the front of the buffer, even if the buffer overflows.

If (radix) is zero, then it is treated as if it were -10; but the leading '+' ordinarily placed for a positive number, is suppressed.

The characters used to represent the number are placed into the buffer, subject to the constraint that at most (n) positions of the buffer may be used.

A trailing '\0' is guaranteed to be written; therefore overflow is indicated by a resulting string length of (n)-1.

Returns: UsbPumpLib_UlongToBuffer () returns the number of byte positions used in the buffer by the result.

Bugs: UsbPumpLib_UlongToBuffer () uses positions at the end of the buffer as temporary storage; hence, all of the buffer must be truly allocated for use by this routine.

If overflow occurs, LEADING digits will be deleted from the number, not trailing digits. However, for signed conversions, the first character will still be a sign character.

If (buf == NULL) or (n == 0), nothing will occur; a 0 will always be returned.

If (n == 1), a '\0' is placed into buf[0].

13.8.13 UsbPumpLib_UlongToBufferHex

Function: Convert the least significant digits of a ULONG into hexadecimal.

Definition:

```
BYTES
UsbPumpLib_UlongToBufferHex(
   TEXT *pBuffer,
   BYTES nBuffer,
   ULONG ulnum,
   BYTES nDigits
   );
```

Description: The low-order nDigits of ulnum is converted to hexadecimal text, and placed into the buffer starting at pBuffer. At most nBuffer – 1 digits will be placed into the buffer (followed by a trailing '\0'). Buffer overflow is signaled by a return of (nBuffer-1) -- note that this case cannot be distinguished from a number whose representation is exactly nBuffer-1 bytes long.

Leading zeros are output as needed.

Returns: Number of bytes placed into the buffer, or 0 to indicate an error.

Notes: If pBuf is NULL, or nBuf is zero, the result is always zero, and nothing else is done.

Otherwise, if nBuf == 1, pBuf[0] is set to '\0', and the result is zero.

See also: UsbPumpLib_UlongToBuffer() is a much more general routine, but there is no limitation to the number of significant digits independent of the buffer size.

13.8.14 UsbPumpLib_InitDeviceControlEp

Function: Perform common initialization for endpoint 0 data structures.

Definition:

```
BOOL UsbPumpLib_InitDeviceControlEp (
    UDEVICE *   pSelf,
    BYTES       BufSize
    );
```

Description: The data structures for endpoint zero for this UDEVICE are allocated and initialized.

Returns: TRUE for success, FALSE otherwise. Failure indicates that the system is out of memory.

13.8.15 UsbPumpLib_CalculateUdeviceSize

Function: Calculate the UDEVICE size from the root table.

Definition:

```
BYTES UsbPumpLib_CalculateUdeviceSize(
        CONST USBRC_ROOTTABLE *     pRoot,
        CONST UDEVICESWITCH *       pSwitch                    );
```

Description: This function calculates UDEVICE structure size from the root table information.

Returns: Number of bytes of UDEVICE structure.

13.8.16 UsbPumpLib_FindAllSizeInfoFromRoot

Function: Find all size information from the root table.

Definition:

```
BYTES UsbPumpLib_FindAllSizeInfoFromRoot (
        CONST USBRC_ROOTTABLE *     pRoot,
        CONST USBPUMP_DEVICE_SIZE_INFO* pInfo
        );
```

Description: This function finds all size information such as how many config, interface set, interface, pipe information from the root table.

Returns: No explicit result.

13.8.17 UsbPumpLib_BeslToMicroSecond

Function: Convert the BESL from the encoded value to microseconds.

Definition:

```
UINT UsbPumpLib_BeslToMicroSecond(
        UINT8  ucBesl
        );
```

Description: The host system communicates to the device the duration of how long the host will drive resume when the host initiates exit from L1 via BESL (Best Effort Service Latency) parameter. The BESL value is a 4-bit encoded value. The encoded value is converted to micro seconds.

Returns: No explicit result.

13.8.18 UsbPumpLib_SHA1_Init

Function: Initialize the SHA1 context.

Definition:

```
USTAT UsbPumpLib_SHA1_Init (
        USBPUMP_SHA1_CONTEXT *      pSha1Ctx,
        USBPUMP_OBJECT_HEADER *     pObjectHeader
        );
```

Description: This function is used for initializing the hash value for SHA-1.

Returns: USTAT_OK if success, otherwise USTAT error code.

13.8.19 UsbPumpLib_SHA1_Update

Function: Hash computation

Definition:

```
USTAT UsbPumpLib_SHA1_Update (
                CONST USBPUMP_SHA1_CONTEXT *    pSha1Ctx,
                CONST UCHAR *   pMsgBuff,
                unsigned long   MsgLength
                );
```

Description: This function computes cumulative message length, intermediate hash value, and residual message length (which is equal to the number of message bits in the last unprocessed block), then converts these message bits into word (32 bits) value. Computed values are updated in SHA1Ctx structure which is used in the next UsbPumpLib_SHA1_Update or in UsbPumpLib_SHA1_Final.

Returns: USTAT_OK if success, otherwise USTAT error code.

13.8.20 UsbPumpLib_SHA1_Final

Function: Final computation

Definition:

```
USTAT UsbPumpLib_SHA1_Final (
        USBPUMP_SHA1_CONTEXT *      pSha1Ctx,
        UCHAR *     pMsgDigest
        );
```

Description: SHA1_Final function is used to obtain the final message digest.

Returns: USTAT_OK if success, otherwise USTAT error code.

13.8.21 UsbPumpLib_PRNG_Initialize

Function: Initialize an instance of pseudo-random number generator.

Definition:

```
VOID UsbPumpLib_PRNG_Initialize (
        USBPUMP_PRNG_CONTEXT *pPrngCtx,
        UINT32 init_x,
        UINT32 init_y,
        UINT32 init_z,
        UINT32 init_c
        );
```

Description: This routine initializes an instance of the pseudo-random number generator. When called with init_x == init_y == init_z == init_c == 0, all values are initialized with the "normal" deterministic initialization values, matching the published reference initialization values given in http://www.cs.ucl.ac.uk/staff/d.jones/GoodPracticeRNG.pdf.

In many cases, use of all zero parameters is suitable, for example when generating a stream of "random" but repeatable values for test purposes. In other cases, preference may be to generate a different sequence each time the generator is used. In these cases, non-zero values must be obtained for at least some of the init_x, init_y, init_z and init_c. A common way to achieve this is to use the time of day. If the user's system provides good sources of entropy-based random numbers, that source may also be used to get 1 to 4 random values.

Returns: No explicit result. The contents of *p are updated in place.

13.8.22 UsbPumpLib_PRNG_NextValue

Function: Generates Pseudo random number based on the seed.

Definition:

```
UINT32 UsbPumpLib_PRNG_NextValue (
        USBPUMP_PRNG_CONTEXT *pPrngCtx
        );
```

Description: This routine generates a pseudo-random UINT32. In order to be reentrant, all context is stored in a user-specified USBPUMP_PRNG_CONTEXT object, which the user must allocate and initialize using UsbPumpLib_PRNG_NextValue().

The PRNG is implemented using G. Marsaglia's KISS generator, as described in http://www.cs.ucl.ac.uk/staff/d.jones/GoodPracticeRNG.pdf, adapted to make it reentrant. The period of the PRNG is around 10^37 -- in other words, the sequence repeats after being called roughly 10^37 times. (Since UINT_MAX is roughly 10^9, a given result will repeat long before the sequence begins to repeat

itself). Because this is a PRNG, the output sequence for a given instance of USBPUMP_PRNG_CONTEXT is deterministic, based on the initial seed value. Therefore, unless it is initialized differently from run-to-run, the same sequence will be realized every time it is used. See UsbPumpLibPrng_Initialize() for information on how to initialize the generator to get variation from run to run. Normal best practice is to use a single USBPUMP_PRNG_CONTEXT object in a given application.

Returns: Pseudo-random value uniformly distributed in the range 0..UINT_MAX - 1.

13.9 Numeric Conversion Routines

To convert a portion of a string to a signed long, call:

```
BYTES UsbPumpLib_BufferToLong(
   CONST TEXT *pBuffer,
   BYTES sizeBuffer,
   UINT base,
   OUT LONG *pResult OPTIONAL,
   OUT BOOL *pfOverflow OPTIONAL
   );
```

To convert a portion of a string to an unsigned long, call:

```
BYTES UsbPumpLib_BufferToUlong(
   CONST TEXT *pBuffer,
   BYTES sizeBuffer,
   UINT base,
   OUT ULONG *pResult OPTIONAL,
   OUT BOOL *pfOverflow OPTIONAL
   );
```

These routines scan up to the first `sizeBuffer` bytes of the string at `pBuffer`, and convert the text into an equivalent `LONG` or `ULONG` value. The argument `base` specifies the radix of the input representation. If 2 <= `base` <= 36, then base is directly used as the radix of the input text. If `base` is 0, then the input radix is determined according to the rules used by standard C: if the number begins with `"0x"` or `"0x"`, it is considered to be base 16; if the number begins with `'0'`, it is considered to be base 8; elsewise, the number is assumed to be decimal.

Prior to converting, these routines skip white space (defined as characters in the ranges 0x1 <= c <= 0x20). They then consume an optional `'-'` sign. (Plus is not permitted.) If base == 0, then a leading `"0x"` or `"0x"` is skipped. Finally, digits are taken from the buffer until all characters have been consumed, or the first non-digit is encountered. A digit that is not legal in the input radix also stops the conversion. (Note that `'\0'` will stop the conversion, by the above rules.)

The primary result of these routines is the number of bytes scanned from the input buffer. The secondary results (`*pResult` and `*pfOverflow`) are guaranteed to be updated, if non-`NULL`.

The results for various conditions are shown in Table 15.

Table 15. Results Matrix for `UsbPumpLib_BufferTo...` Routines

Condition	Result	*pResult	*pfOverflow
No valid number seen	0	0	FALSE
Valid number seen	Number of bytes scanned	Converted number	FALSE
Number valid, but too large or small for result	Number of bytes scanned	LONG_MIN, LONG_MAX or ULONG_MAX, as appropriate	TRUE

14. Basic HIL Functions

The routines listed in this section are required to be present in all Hardware Interface Layer ports. They are provided by MCCI as part of the HIL porting process. For a new port, some of these functions may need to be modified.

14.1 Low Level Platform Interface

14.1.1 UPF_GetEventContext

Function: Library form of (obsolescent) UPF_GetEventContext ().

Definition:

```
PUEVENTCONTEXT UPF_GetEventContext(
            UPLATFORM *pPlatform
            );
```

Description: The event context field of the UPLATFORM is extracted.

Returns: The context pointer, or NULL if none is set.

Notes: This routine is named UPF_xx for historical reasons. (The UPLATFORM is now a portable structure, and so UHIL_ would perhaps be a more suitable prefix).

This routine is frequently implemented as a macro, implying that the arguments may be evaluated more than once. To force a subroutine call, write:

result = (UPF_GetEventContext)(pPlatform);

The parentheses around the function name will defeat the macro pre-processor.

14.1.2 UHIL_Stop

Function: Downcall to the HIL from a device, saying that the device wants to stop.

Definition:

```
VOID UHIL_Stop (
    UPLATFORM *platform,
    UDEVICE *dev
    );
```

Description: When a device is removed (e.g., a PCMCIA card), the DataPump detects it. At that time, the DataPump calls UHIL_Stop() to indicate a device removal is occurring This routine should do whatever is needed to assist the removal: unmapping resources, releasing IRQs, etc. `platform` corresponds to a PDO in WDM lingo, and represents what was previously connected; the `dev` is passed down so the platform code has some idea of what it is connected to without having knowledge of the contents of the UDEVICE.

Returns: Nothing.

Notes: Unlike WDM, unexpected removals are the norm. So, failure is not an option for this routine. The classic removal sequence is:

- removal detection synchronizes;
- removal detection calls `dev->udevsw_StopDevice`;
- the StopDevice routine calls UHIL_Stop at the appropriate moment so that UHIL_Stop gets control and shuts down the physical device.

The 'removal detection' module is not supplied with the standard distribution.

The `platform` is redundant, because it can be fetched from the UDEVICE.

Porting: This routine must be rewritten to match the user's OS/platform's removal discipline (if any). If rewritten, it must be moved to another directory, because the original of the file will otherwise get updated with each release.

14.2 Initialization

14.2.1 UHIL_InitVars

Function: Initialize BSS variables.

Definition:

```
VOID UHIL_InitVars (VOID);
```

Description: This routine is responsible for initializing our Block Started by Symbol (BSS) variables at the beginning of time. STARTUP CODE MUST CALL THIS ROUTINE BEFORE CALLING ANY OTHER.

Returns: Nothing.

14.3 Low Level Services

The functions described in the section are primarily for use by Application programs. They provide the application with a device-independent set of "system services".

14.3.1 Interrupts

14.3.1.1 New version: UHIL_INTERRUPT_SYSTEM_INTERFACE

14.3.1.2 UHIL_OpenInterruptConnection

Function: Opens an interrupt connection on a specific interrupt number.

Environment: Supervisor mode.

Definition:

```
UHIL_INTERRUPT_CONNECTION_HANDLE UHIL_OpenInterrupConnection {
       CONST UHIL_INTERRUPT_SYSTEM_INTERFACE  *pInterruptInterface,
       UHIL_INTERRUPT_RESOURCE_HANDLE hResource
       );
```

Description: The open interrupt connection is called to start communication with the interrupt handling system. The function returns an opaque handle, which enables the start of managing a particular IRQ, abstracted by the resource handle argument. The IRQ is placed in a quieted state.

Returns A handle to the interrupt connection.

14.3.1.3 UHIL_CloseInterruptConnection

Function: Closes an interrupt connection on a specific interrupt number.

Environment: Supervisor mode.

Definition:

```
BOOL UHIL_CloseInterruptConnection (
        UHIL_INTERRUPT_CONNECTION_HANDLE hConnection
        );
```

Description: Platforms that support plug and play or device shutdown (e.g, the DOS programs which want to unhook and exit) also have to provide a close-interrupt-handle primitive. This can be stubbed on embedded platforms that have no use for this.

Returns: TRUE if successful, FALSE otherwise.

14.3.1.4 UHIL_ConnectToInterrupt

Function: Connects an ISR to an interrupt.

Environment: Supervisor mode.

Definition:

```
BOOL UHIL_ConnectToInterrupt (
        UHIL_INTERRUPT_CONNECTION_HANDLE hConnection,
        UHIL_INTERRUPT_SERVICE_ROUTINE *pIsrFunction,
        VOID *pIsrContext
        );
```

Description: The connect-to-interrupt function is called to connect an ISR to an interrupt. If the interrupt system permits, the interrupt is initially disabled at the interrupt controller.

Returns: TRUE if successful, FALSE otherwise.

14.3.1.5 UHIL_DisconnectFromInterrupt

Function: Disconnects an ISR without releasing the interrupt handle.

Environment: Supervisor mode.

Definition:

```
BOOL UHIL_DisconnectFromInterrupt (
        UHIL_INTERRUPT_CONNECTION_HANDLE hConnection
        );
```

Description: Platforms that support power managemetn or dynamic USB hardware shutdown might have to implement the disconnect-from-interrupt primitive. This can be stubbed on embedded platforms that have no use for this.

Returns: TRUE if successful, FALSE otherwise.

14.3.1.6 UHIL_InterruptControl

Function: Used for manipulating the interrupt subsystem.

Environment: Supervisor mode.

Definition:

```
BOOL UHIL_InterruptControl(
       UHIL_INTERRUPT_CONNECTION_HANDLE hConnection,
       BOOL fEnable
       );
```

Description: Enable or disable the interrupt identified by hConnection. The user should be aware of platforms on which this cannot be implemented. Conversely, there may be DCDs that absolutely require this.

Returns: TRUE if successful, FALSE otherwise.

14.3.1.7 UHIL_ ConnectToInterruptV2

Function: Connect ISRs (line and message ISR) to an interrupt.

Environment: Supervisor mode.

Definition:

```
BOOL UHIL_ConnectToInterruptV2 (
       UHIL_INTERRUPT_SYSTEM_INTERFACE,
       UHIL_INTERRUPT_CONNECTION_HANDLE,
       VOID *,
       UHIL_LINE_INTERRUPT_SERVICE_ROUTINE,
       UHIL_MESSAGE_INTERRUPT_SERVICE_ROUTINE,
       UHIL_MP_LINE_INTERRUPT_SERVICE_ROUTINE,
       UHIL_MP_MESSAGE_INTERRUPT_SERVICE_ROUTINE
       );
```

Description: The connect-to-interrupt function is called to connect an ISR to an interrupt. At most one of the provided ISRs will be used on any given system. The caller need not provide all four routines to this service, but if the system needs to use a routine that is not provided, the connection will not be made.

In most cases, the platform decides in advance how an interrupt needs to be connected. Furthermore, uniprocessor-model interrupt processing might not be available on a given platform. For maximum portability to new systems, DCDs and HCDs should provide pMpLineIsr and pMpMessage ISR.

If the interrupt system permits, the interrupt is initially disabled at the interrupt controller.

Returns: TRUE if successful, FALSE otherwise.

14.3.1.8 UHIL_ InterruptSystemIOCTL

Function: Get information about the interrupt represented by an interrupt connection handle.

Environment: Supervisor mode.

Definition:

```
USBPUMP_IOCTL_RESULT UHIL_InterruptSystemIOCTL(
       UHIL_INTERRUPT_SYSTEM_INTERFACE,
       USBPUMP_IOCTL_CODE,
       CONST VOID *,
       VOID *
       );
```

Description: UHIL_InterruptConnectionIOCTL allows users to get information about the implementation and requirements from the interrupt system for a given interrupt handle. It may be called after opening an interrupt connection handle, and can return the following information:

- Whether the interrupt will connect in line mode or message mode.
- If in message mode, the number of messages that can be handled by the interrupt system.
- Whether the interrupt system supports uniprocessor-model ISRs.
- Whether the interrupt system supports multiprocessor-model ISRs.

Returns: UHIL_ InterruptSystemIOCTL returns the usual DataPump IOCTL result codes.

14.3.2 UHIL_PostCallback

Function: Post a callback event for deferred processing.

Definition:

```
VOID UHIL_PostCallback (
    UPLATFORM *pPlatform,
    CALLBACKCOMPLETION *ctx
    )
```

Description: This function allows the caller to queue a callback into the platform's event queue. A caller can use this event to defer processing. The CALLBACKCOMPLETION is a derivable structure with the following base class elements:

callback_pfn: This is the pointer to the callback function. This function takes one argument, the value of the callback_ctx.

callback_ctx: This value is the caller-specified context pointer or data. It is the only argument supplied to the function callback_pfn.

Returns: Nothing.

14.3.3 UHIL_SetFirmwarePoll

Function: Establish function to be called for polling.

Definition:

```
FIRMWAREPOLLFN *UHIL_SetFirmwarePoll (
        PUPOLLCONTEXT   pc,
        FIRMWAREPOLLFN  *newfn,
        VOID            *ctx
        );
```

Description: UHIL_SetFirmwarePoll() arranges for the specified C function to be called as part of the event queue processing in the background. This routine is called just before checking the event queue, and so can be used for non-interrupt-driven polling of devices, status checks, and so forth.

Returns: The last poll function.

14.4 Kernel Services

The HIL supports a general set of kernel-level services that must be present in any port. This allows firmware applications and HIL ports to be constructed in a platform independent means.

14.4.1 UHIL_di

Function: Disable interrupts.

Definition:

```
CPUFLAGS UHIL_di (VOID);
```

Description: The CPU status word is modified so that interrupts are disabled.

Returns: A platform-specific representation of the CPU status word before it was modified. It is assumed that the caller will save this value and use it as the argument to UHIL_setpsw () to restore the previous value. Portable code must treat this as opaque data. Note that MCCI's code disables interrupts sparingly,

and does not attempt to take advantage of the multiple priority levels available on some CPUs.

14.4.2 UHIL_setpsw

Function: Restore interrupts.

Definition:

```
VOID UHIL_setpsw (
       ARG_CPUFLAGS psw
       );
```

Description: The argument `psw` is written into the CPU status word to restore interrupts to their previous value. Users of this function should never make any assumptions about the meanings of various bits in CPUFLAGS or ARG_CPUFLAGS.

Returns: Nothing.

14.4.3 UHIL_swc

Function: Software Check.

Definition:

```
VOID UHIL_swc (
       UHIL_SWCCODE code
       );
```

Description: This function is used by firmware to record catastrophic failures. This call must be implemented to either stop execution (for debug platforms) or to attempt recovery and restart of the device.

Returns: This call never returns.

14.4.4 UHIL_FindDescriptor

Function: Look up the requested descriptor.

Definition:

```
   UHILSTATUS UHIL_FindDescriptor (
           CONST USBRC_ROOTTABLE * CONST pRoot,     -- IN root of table
           ARG_USHORT  type,                        -- IN descriptor type
           ARG_USHORT  index,                       -- IN descriptor index
           ARG_USHORT  lid,                         -- IN descriptor language
id
           BYTES       *psize,   -- OUT descriptor size
```

```
        UCHAR       **ppDesc  -- OUT descriptor
        );
```

Description: `index`, `type`, and `lid` are usedto index into the descriptor table and return the requested descriptor.

Returns: UHILERR_SUCCESS: descriptor found, `psize` and `ppDesc` are valid.
UHILERR_BAD_DTYPE: bad descriptor type requested
UHILERR_BAD_DINDEX: bad descriptor index requested
UHILERR_BAD_DLID: bad descriptor language ID requested.

14.4.5 UHILSTATUS UHIL_FindDescriptorV2

Function: Look up the requested descriptor.

Definition:

```
    UHILSTATUS UHIL_FindDescriptorV2(
            CONST USBRC_ROOTTABLE * CONST pRoot,     -- IN root of table
            USBPUMP_DEVICE_SPEED bCurSpeed, -- IN current device speed
            USBPUMP_DEVICE_SPEED_MASK SupportedSpeeds, -- IN supported speeds
map
            ARG_USHORT  type,                        -- IN descriptor type
            ARG_USHORT  index,                       -- IN descriptor index
            ARG_USHORT  lid,                    -- IN descriptor language id
            BYTES       *psize,   -- OUT descriptor size
            UCHAR       **ppDesc  -- OUT descriptor
            );
```

Description: SupportedSpeeds and bCurSpeed are used to get the requested speed. With the requested speed (bOtherSpeed/bCurSpeed), index, type, and lid are used to index into the descriptor table and return the requested descriptor.

For the string descriptor, there is no speed information needed. index, type, and lid are used to index into the descriptor table and return the requested descriptor.

Returns: UHILERR_SUCCESS: descriptor found, `psize` and `ppDesc` are valid.
UHILERR_BAD_DTYPE: bad descriptor type requested
UHILERR_BAD_DINDEX: bad descriptor index requested
UHILERR_BAD_DLID: bad descriptor language ID requested.

14.4.6 UHIL_le_getuint16

Function: Extract 2-byte value encoded in little-endian form, no alignment restrictions.

Definition:

```
UINT16 UHIL_le_getuint16 (
    CONST UINT8 *buf
    );
```

Description: The 2 bytes starting at `buf` are converted into a native UINT16.

Slightly different than nativedat() because OK alignment is not assumed.

Returns: The value.

Macro for invoking:

```
UHIL_LE_GETUINT16 (buf)
```

14.4.7 UHIL_le_getuint32

Function: Extract 4-byte value encoded in little-endian form, no alignment restrictions.

Definition:

```
UINT32 UHIL_le_getuint32 (
    CONST UINT8 *buf
    );
```

Description: The 4 bytes starting at `buf` are converted into a native UINT32.

Slightly different than nativedat() because OK alignment is not assumed.

Returns: The value.

Macro for invoking:

```
UHIL_LE_GETUINT32 (buf)
```

14.4.8 UHIL_le_getuint64

Function: Extract 8-byte value encoded in little-endian form, no alignment restrictions.

Definition:

```
UINT64 UHIL_le_getuint64(
    CONST UINT8 *buf
    );
```

Description: The 8bytes starting at buf are converted into a native UINT64.

Slightly different than nativedat() because OK alignment is not assumed.

Returns: The value.

Macro for invoking:

```
UHIL_LE_GETUINT64(buf)
```

14.4.9 UHIL_le_getuint128_s

Function: Convert a little-endian string of 16 bytes into a UINT128_S.

Definition:

```
VOID UHIL_le_getuint128_s(
    UINT128_S *pResult,
    CONST UINT8 *pBuffer
    );
```

Description: The 16 bytes at pBuffer are converted in to a native-format UINT128_S.

Slightly different than nativedat() because OK alignment is not assumed.

Returns: No explicit result; *pResult is updated in place.

Macro for invoking:

```
UHIL_LE_GETUINT128_S(pResult, pBuffer)
```

14.4.10 UHIL_le_getint128_s

Function: Convert a little-endian string of 16 bytes into an INT128_S.

Definition:

```
VOID UHIL_le_getint128_s(
    INT128_S *pResult,
    CONST INT8 *pBuffer
    );
```

Description: The 16 bytes at pBuffer are converted in to a native-format INT128_S.

Since most compilers don't handle 128-bit integers natively, the value is broken up into two 64-bit values.

Returns: No explicit result; *pResult is updated in place.

Macro for invoking:

```
UHIL_LE_GETINT128_S(pResult, pBuffer)
```

14.4.11 UHIL_le_putuint16

Function: Convert a 16-bit value into 2 bytes, and store in buffer.

Definition:

```
VOID UHIL_le_putuint16 (
    UINT8 *buf,  -- where to put it,
    UINT16 val   -- what to store.
    )
```

Description: The value is cracked into 2 bytes, then stored.

Returns: Nothing.

Macro for invoking:

```
UHIL_LE_PUTUINT16 (buf, val)
```

14.4.12 UHIL_le_putuint32

Function: Convert a 32-bit value into 4 bytes, and store in buffer.

Definition:

```
VOID UHIL_le_putuint32 (
    UINT8 *buf,  -- where to put it,
    UINT32 val   -- what to store.
    )
```

Description: The value is cracked into 4 bytes, then stored.

Returns: Nothing.

Macro for invoking:

```
UHIL_LE_PUTUINT32 (buf, val)
```

14.4.13 UHIL_be_getuint16

Function: Extract 2-byte value encoded in big-endian form, no alignment restrictions.

Definition:

```
UINT16 UHIL_be_getuint16(
    CONST UINT8 *buf
    );
```

Description: The 2 bytes starting at buf are converted into a native UINT16.

Slightly different than nativedat() because OK alignment is not assumed.

Returns: The value.

Macro for invoking:

```
UHIL_BE_GETUINT16(buf)
```

14.4.14 UHIL_be_getuint32

Function: Extract 4-byte value encoded in big-endian form, no alignment restrictions.

Definition:

```
UINT16 UHIL_be_getuint32(
    CONST UINT8 *buf
    );
```

Description: The 4 bytes starting at buf are converted into a native UINT32.

Slightly different than nativedat() because OK alignment is not assumed.

Returns: The value.

Macro for invoking:

```
UHIL_BE_GETUINT32(buf)
```

14.4.15 UHIL_be_getuint64

Function: Extract 8-byte value encoded in big-endian form, no alignment restrictions.

Definition:

```
UINT16 UHIL_be_getuint64(
    CONST UINT8 *buf
    );
```

Description: The 8 bytes starting at buf are converted into a native UINT64.

Slightly different than nativedat() because OK alignment is not assumed.

Returns: The value.

Macro for invoking:

```
UHIL_BE_GETUINT64(buf)
```

14.4.16 UHIL_be_getint128_s

Function: Convert a big-endian string of 16 bytes into an INT128_S.

Definition:

```
VOID UHIL_be_getint128_s(
    INT128_S *pResult,
    CONST UINT8 *pBuffer
    );
```

Description: The 16 bytes at pBuffer are converted into a native format INT128_S.

Since most compilers don't handle 128-bit integers natively, the value is broken up into two 64-bit values.

Returns: No explicit result; *pResult is updated in place.

Macro for invoking:

```
UHIL_BE_GETINT128_S(pResult, pBuffer)
```

14.4.17 UHIL_be_getuint128_s

Function: Convert a big-endian string of 16 bytes into a UINT128_S.

Definition:

```
VOID UHIL_be_getuint128_s(
    UINT128_S *pResult,
    CONST UINT8 *pBuffer
    );
```

Description: The 16 bytes at pBuffer are converted into a native format UINT128_S.

Since most compilers don't handle 128-bit integers natively, the value is broken up into two 64-bit values.

Returns: No explicit result; *pResult is updated in place.

Macro for invoking:

```
UHIL_BE_GETUINT128_S(pResult, pBuffer)
```

14.4.18 UHIL_be_putuint16

Function: Convert a 16-bit value into 2 bytes, and store in buffer.

Definition:

```
VOID UHIL_be_putuint16(
    UINT8 *buf, -- where to put it,
    UINT16 val  -- what to store.
    );
```

Description: The value is cracked into 2 bytes, then stored.

Returns: Nothing.

Macro for invoking:

```
UHIL_BE_PUTUINT16(buf, val)
```

14.4.19 UHIL_be_putuint32

Function: Convert a 32-bit value into 4 bytes, and store in buffer.

Definition:

```
VOID UHIL_be_putuint32(
    UINT8 *buf,  -- where to put it,
    UINT32 val   -- what to store.
    );
```

Description: The value is cracked into 4 bytes, then stored.

Returns: Nothing.

Macro for invoking:

```
UHIL_BE_PUTUINT32(buf, val)
```

14.4.20 UHIL_be_putuint64

Function: Convert a 64-bit value into 8 bytes, and store in buffer, in big-endian form.

Definition:

```
VOID UHIL_be_putuint64(
    UINT8 *buf,  -- where to put it,
    UINT64 val   -- what to store.
    );
```

Description: The value is cracked into 8 bytes, then stored, from most significant to least.

Returns: Nothing.

Macro for invoking:

```
UHIL_BE_PUTUINT64(buf, val)
```

14.4.21 UHIL_be_putint128_s

Function: Encode an INT128_S into big-endian wire format in a 16-byte buffer.

Definition:

```
VOID UHIL_be_putint128_s(
    UINT8 *pBuf,
```

```
    CONST INT128_S *pValue
    );
```

Description: The value is simply broken down into bytes and is written to the buffer. Since most compilers don't support 128 bits natively, the value is broken into two 64-bit values first.

Returns: No explicit result.

Macro for invoking:

```
UHIL_BE_PUTINT128_S(pBuf, pValue)
```

14.4.22 UHIL_be_putuint128_s

Function: Encode an UINT128_S into big-endian wire format in a 16-byte buffer.

Definition:

```
VOID UHIL_be_putuint128_s(
    UINT8 *pBuf,
    CONST UINT128_S *pValue
    );
```

Description: The value is simply broken down into bytes and is written to the buffer. Since most compilers don't support 128 bits natively, the value is broken into two 64-bit values first.

Returns: Nothing.

Macro for invoking:

UHIL_BE_PUTUINT128_S(pBuf, pValue)

14.4.23 UHIL_udiv64

Function: Divide 64-bit variable

Definition:

```
UINT64 UHIL_udiv64 (
    UINT64 dividend,
    UINT64 divisor
    );
```

Description: This is default implementation of 64-bit division function.

Returns: Quotient value of division.

Macro for invoking:

UHIL_UDIV64(dividend, divisor);

14.4.24 UHIL_urem64

Function: Divide 64-bit variable and return remainder

Definition:

```
UINT64 UHIL_urem64(
    UINT64  dividend,
    UINT64  divisor
    );
```

Description: This is default implementation of 64-bit remainder function. Divide 64-bit variable and return remainder

Returns: Remainder value of division.

Macro for invoking:

UHIL_UREM64 (dividend, divisor);

14.5 Library Functions

The following functions are provided for convenience.

14.5.1 UHIL_cpybuf

Function: Copy a buffer.

Definition:

```
BYTES UHIL_cpybuf (
       VOID *dest,
       CONST VOID *src,
       BYTES bufsz
       );
```

Description: The contents of the source buffer are copied into the destination buffer. If either pointer is NULL, no data is copied.

Returns: `bufsz`.

14.5.2 UHIL_lenstr

Function: The MCCI-style strlen () that is a total-function of its input.

Definition:

```
BYTES UHIL_lenstr (
       CONST TEXT *pString
       );
```

Description: UHIL_lenstr() returns the length of a NULL-terminated string.

It differs from strlen() in the following ways:

1. It is always available.

2. Its result is always unsigned (BYTES), rather than size_t.

3. It is a total function of the set of {pointers, NULL}.

UHIL_lenstr (NULL) is zero; whereas strlen(NULL) is undefined.

Returns: Number of bytes in the input string.

14.5.3 UHIL_cpynstr

Function: A portable string copy routine: copies a bunch of strings, stopping when output string exhausted or when all strings copied.

Definition:

```
BYTES UHIL_cpynstr (b, n, p1, p2, ..., NULL)

        TEXT *b;output buffer
        BYTES n;size of b
        TEXT *p1, ...   NULL-terminated list of input strings;
```

Description: UHIL_cpynstr () copies each string, checking to make sure that the buffer isn't overrun. The last item in the list must be a 'NULL' pointer. If there is room, a '\0' is placed at the end of the buffer.

Returns: UHIL_cpynstr () returns the number of bytes actually placed in b, which will be in [0, n-1]. This does NOT include the trailing '\0', which is always appended if n>0 and b != NULL. If the result is n-1, then the buffer may have overflowed. Generally, one wants to interpret this case as buffer overflow, anyway.

14.5.4 UHIL_cmpbuf

Function: A portable buffer comparison routine with known semantics.

Definition:

```
BOOL UHIL_cmpbuf (
        CONST VOID *p1,
        CONST VOID *p2,
        BYTES bufsize
        );
```

Description: The buffer at p1 is compared to the buffer at p2; each buffer is assumed to be bufsize bytes long. If p1 is equal to p2, then the result is TRUE. Otherwise if either p1 or p2 is NULL, then the result is FALSE. Otherwise if bufsize is zero, then the result is TRUE. Otherwise the result depends on the byte-by-byte comparison of the buffer.

Returns: TRUE if the buffers compare equal, FALSE otherwise.

14.5.5 UHIL_cmpstr

Function: A portable string comparison routine with known semantics.

Definition:

```
BOOL UHIL_cmpstr (
        CONST TEXT *p1,
        CONST TEXT *p2
        );
```

Description: The buffer at p1 is compared to the buffer at p2; each buffer is assumed to be bufsize bytes long. If p1 is equal to p2, then the result is TRUE. Otherwise if either p1 or p2 is NULL, then the result is FALSE. Otherwise if bufsize is zero, then the result is TRUE. Otherwise the result depends on the byte-by-byte comparison of the buffer.

Returns: TRUE if the buffers compare equal, FALSE otherwise.

14.5.6 UHIL_fill

Function: Fill a buffer.

Definition:

```
BYTES UHIL_fill (
        VOID *buffer,
        BYTES bufsz,
        ARG_UCHAR value
        );
```

Description: The contents of the `buffer` are filled with `value`. Returns immediately if `buffer` is NULL.

Returns: The count of bytes written into the buffer (the buffer size.)

14.6 Debug Logging Functions

The HIL layer must supply a consistent set of console/debug output routines. These routines are normally coded using a large memory buffer and print-behind that is driven from the idle loop, so that calls to these routines will not affect the run-time performance of the USB DataPump.

14.6.1 UHIL_DebugPrintEnable

Function: Enable or disable debug printing.

Definition:

```
BOOL UHIL_DebugPrintEnable (
    UPLATFORM *pPlatform,
    BOOL enable
    );
```

Description: The printing enable state is set to what the caller asks for and return the previous setting.

Returns: The previous enable state.

14.6.2 UHIL_FlushChar

Function: Flush the print queue.

Definition:

```
VOID UHIL_FlushChar (VOID);
```

Description: The output function is called repeatedly until the queue is empty.

Returns: Nothing.

Notes: OBSOLETE: please use UsbDebug_PlatformFlush(). Function UHIL_FlushChar is deprecated in favor of UsbDebug_PlatformFlush, and is only provided for backward compatibility.

14.6.3 UHIL_PrintChar

Function: Print a single character.

Definition:

```
VOID UHIL_PrintChar(
    UPLATFORM *pPlatform,
    ARG_CHAR ch
    );
```

Description: The character is checked to see if it's a LF. If it is, the full CRLF sequence is printed, otherwise, just the character is printed.

Returns: Nothing.

14.6.4 UHIL_PrintCharTransparent

Function: Print the character without translation.

Definition:

```
VOID UHIL_PrintCharTransparent(
    UPLATFORM *pPlatform,
    ARG_CHAR ch
    );
```

Description: The character is added to the queue. If the queue is getting full, some chars are attempted to be pushed out through the out function.

Returns: Nothing.

Notes: This is merely a trampoline routine that connects to the actual debug module.

14.6.5 UHIL_PrintCrlf

Function: Print a carriage return, linefeed sequence.

Definition:

```
VOID UHIL_PrintCrlf(
    UPLATFORM *pPlatform
    );
```

Description: This function calls the print char routine to print an LF char. The translator in that routine will see the LF and put out a separate CR before the LF.

Returns: Nothing.

14.6.6 UHIL_PrintInit

Function: Initialize UHIL Output services.

Definition:

```
VOID UHIL_PrintInit (
    UPLATFORM *pPlatform,
    PDBGOUTFN outfn     -- output fn to drain printq
    );
```

Description: This functin initializes the queue pointers. If the `outfn` ptr supplied is null, the internal 'sink' routine is hooked up to drain the print queue. Otherwise, `outfn` will be used for this service.

Returns: Nothing.

14.6.7 UHIL_PrintPoll

Function: Poll the output routine.

Definition:

```
VOID UHIL_PrintPoll(
    UPLATFORM *pPlatform
    );
```

Description: The output function is called.

Returns: Nothing.

14.6.8 UHIL_PrintStr

Function: Print a string (ASCIIZ).

Definition:

```
VOID UHIL_PrintStr (CONST TEXT *pstr);
```

Description: This calls the print char routine for each successive character in the string until the zero-terminator is hit.

Returns: Nothing.

14.6.9 UHIL_PrivateSink

Function: Drain the printq.

Definition:

```
VOID UHIL_PrivateSink (VOID);
```

Description: If a next byte exists in the printq, remove it.

Returns: Nothing.

14.6.10 UHIL_PrintUnicodeStr

Function: Print a unicode string.

Definition:

```
VOID UHIL_PrintUnicodeStr (
    UPLATFORM *pPlatform,
    CONST VOID *pstr,
    ARG_USHORT count
    );
```

Description: The print char routine is called for each successive character in the string until the zero-terminator is hit.

Returns: Nothing.

14.6.11 UHIL_PrintUshort

Function: Print a 4-digit hex value.

Definition:

```
VOID UHIL_PrintUshort(
    UPLATFORM *pPlatform,
    ARG_USHORT us
    );
```

Description: This function converts each nibble of `us` into an ASCII Hex character. This is done high->low and each character is printed as it is translated.

Returns: Nothing.

14.6.12 UHIL_PQCheckChar

Function: Check to see if any data is waiting in the print queue.

Definition:

```
BOOL UHIL_PQCheckChar(
    UPLATFORM *pPlatform
    );
```

Description: This compares the queue head and tail for equality.

Returns: TRUE: data waiting, FALSE: queue empty.

14.6.13 UHIL_PQGetChar

Function: Get the next queued character from the print queue.

Definition:

```
TEXT UHIL_PQGetChar(
    UPLATFORM *pPlatform
    );
```

Description: If no characters are waiting, 0 is returned. Otherwise, the next character is removed from the queue and the head pointer is advanced.

Returns: The next queued character from print queue.

14.6.14 UHIL_PrintBuf_Default

Function: Print a buffer, with newline translation.

Definition:

```
UPLATFORM_DEBUG_PRINT_BUF_FN UHIL_PrintBuf_Default;

VOID UHIL_PrintBuf_Default(
    UPLATFORM *pPlatform,
    CONST TEXT *pBuf,
    BYTES nBytes
    );
```

Description: Each character of the buffer is displayed. '\n' sequences are expanded as appropriate.

Returns: No explicit result.

Notes: This function can be used with any PrintBufTransparent method, if simply \n => \r\n translation is required

14.6.15 UHIL_PrintBufTransparent_Default

Function: Print the character without translation.

Definition:

```
UPLATFORM_DEBUG_PRINT_BUF_FN UHIL_PrintBufTransparent_Default;

VOID UHIL_PrintBufTransparent_Default (
    UPLATFORM *pPlatform,
    CONST TEXT *pBuf,
    BYTES nBuf
    );
```

Description: The character is added to the queue. If the queue is getting full, some chars are pushed out through the out function.

Returns: None.

Notes: No need to disable ints here as long as this is guarded by the datapump interlocks.

14.6.16 UHIL_FlushChar_Default

Function: Flush the printq.

Definition:

```
UPLATFORM_DEBUG_FLUSH_FN UHIL_FlushChar_Default;
```

```
VOID UHIL_FlushChar_Default (
    UPLATFORM *pPlatform
    );
```

Description: The output function is repeatedly called until the queue is empty.

Returns: None.

14.6.17 UHIL_PrintPoll_Default

Function: Poll the output routine.

Definition:

```
UPLATFORM_PRINT_POLL_FN  UHIL_PrintPoll_Default;

VOID UHIL_PrintPoll_Default(
    UPLATFORM *pPlatform
    );
```

Description: The output function is called.

Returns: None.

14.6.18 UHIL_DebugPrintEnable_Default

Function: Enable or disable debug printing.

Definition:

```
UPLATFORM_DEBUG_PRINT_ENABLE_FN UHIL_DebugPrintEnable_Default;

BOOL UHIL_DebugPrintEnable_Default(
    UPLATFORM *pPlatform,
    BOOL enable
    );
```

Description: This function sets the printing enable state to what the caller asks for and returns the previous setting.

Returns: None.

15. Build System Reference

This chapter gives some simple examples of how to build MCCI DataPump. Refer to 950000289-(MCCI-USB-DataPump-Build-Process-and-Build-Plan) for the details.

15.1 Creating a build tree

This section talks about build trees and present makebuildtree reference info.

When building the DataPump, all work happens "outside" the source code tree, in a "build" tree. Each build tree has a "pre-set" collection of settings: CPU, compiler, OS, target board and compile-time options. In MCCI building system, "MCCI port" is the same as "target board" or "BSP".

To set up a build tree, "makebuildtree" command is used It lives in usbkern by default.

15.1.1 Running makebuildtree on Windows with Thompson Toolkit

The Thompson Toolkit is MCCI's reference build environment. If using the Thompson Toolkit, makebuildtree.sh can be invoked directly:

```
cd path/usbkern
./makebuildtree.sh [options] portname
```

For example, command "makebuildtree catena1650" is to set up a checked build in "i386/catena1650/w32-microsoft-checked".

Command "makebuildtree -b free catena1610" is to set up a free build in "i386/catena1610/w32-microsoft-free".

15.1.2 Running makebuildtree on Windows with Cygwin

If using the Korn shell installed on Cygwin, `makebuildtree.sh` can be invoked directly by using the following command:

```
cd path/usbkern
./makebuildtree.sh [options] portname
```

Otherwise, use bash, as follows:

```
cd path/usbkern
bash makebuildtree.sh [options] portname
```

15.1.3 Running makebuildtree on Unix-based systems (Linux, Solaris, NetBSD)

If building on Unix-based systems, the Korn shell must be installed. Invoke `makebuildtree.sh` directly, using the following command.

```
cd path/usbkern
./makebuildtree.sh [options] portname
```

Note that the MCCI `tools/bin` directory must be among the directories listed in the `PATH` environment variable.

15.2 Building

After build tree is set up, go to the build directory and run "bsdmake".

15.2.1 Building on Windows with Thompson Toolkit

A sample command for building with VC6.0 is:

```
MSVC60_HOME='c:/progra~1/micros~4/vc98' \
   PATH="c:\progra~1\micros~4\common\msdev98\bin;$PATH" \
   TOOLS='c:/mcci/tools' \
       bsdmake
```

15.2.2 Building on Windows with Cygwin

A sample command for building with VC6.0 using bash as the build shell is:

```
MSVC60_HOME='c:/progra~1/micros~4/vc98' \
   PATH="/cygdrive/c/progra~1/micros~4/common/msdev98/bin:$PATH" \
   TOOLS='c:/mcci/tools' \
   MAKESHELL='c:\cygwin\bin\bash.exe' \
       bsdmake
```

Note that KSH may also be used:

```
MSVC60_HOME='c:/progra~1/micros~4/vc98' \
   PATH="/cygdrive/c/progra~1/micros~4/common/msdev98/bin:$PATH" \
   TOOLS='c:/mcci/tools' \
   MAKESHELL='c:\cygwin\bin\pdksh.exe' \
       bsdmake
```

In either case, the first two lines provide the necessary setup for using MSVC 6.0. The next line ("TOOLS=…") provides the pointer to the top level of the MCCI tools directory. The final line tells `bsdmake` which shell to use for building.

`bash` may be used as the build shell even if the command line shell is something else (`csh`, `tcsh`, `pdksh`, etc.) The same is true for `pdksh`.

15.3 Build System Q&A

15.3.1 Supporting Customer Development Environments

My project has its own build system that I must use, and we have a source license for the DataPump. How can we integrate your build system with ours?

MCCI does not support building the DataPump outside our build environment. Therefore, you must divide the build process into two steps: first build the DataPump link libraries from source, then build your application. The supported development flow is:

3. Build the DataPump libraries using the MCCI build environment

4. Optionally, copy the newly created DataPump libraries to a standard place

5. Build the target application using the customer build environment, linking against the libraries built during the first step, and using the DataPump header files. This normally requires running `usbrc` from the customer build environment.

This flow has the advantage that the DataPump code need not be built every time you build your application. Even if nothing has changed, dependency checking for the DataPump may be time consuming.

Optionally, you can arrange for the DataPump to create your application as a library inside the DataPump build system, and then link against that library. For some target operating systems, this approach is the default. On others, you can simply create a `UsbMakefile.inc` that specifies your URC file but uses LIBRARY= to specify a target library rather than using PROGRAM= to specify a target program.

Appendix A DataPump Supported Configurations

The MCCI USB DataPump Software Development kit was designed to support a large number of combinations of Target CPU, Target USB interface, Target Operating System, and Cross Compilation Development environment. Table A-1 lists the currently supported combinations. However, not all combinations are directly supported with the product. If you desire to use a combination not listed in Table A-1, please contact MCCI.

Table A-1. Supported Hardware and Development Tools

Target CPU	Target USB Interface	Target Operating System	Cross Compilation Development Tools
ARM ARC Tensilica R3000 FT32 68K, ColdFire M-CORE PPC RX Sparc X86 X86_64	Agere USS820 NXP D12, ISP1362, ISP1582, ISP1583, ISP1761, IP5213, IP5220, IP9028 Epson S1R72005 FTDI FT900 TI OMAP 5912, 1610, etc Freescale i.MX1, i.MXL, i.MX21, i.MX6 IP from Synopsys, Mentor, Cadence, Renesas Synopsys XDCI	None (straight code) OSE ThreadX Nucleus Linux INTEGRITY eCos Micrium uCos-II,-III uITRON	GNU Clang Diab IAR MetaWare Microsoft C Green Hills Arm ADS, SDT, RealView and DS-5 Keil

Appendix B DataPump Directory Structure Overview

Since many types of applications, processors, DCDs, protocols, etc. are covered with the MCCI USB DataPump software, an overview of the software provided is very helpful. The following table is a description of the contents of the main directories in the DataPump installation (begins under the MCCI/ directory).

Table B-1. DataPump Directory and its Purpose

Directory Name (under MCCI/ directory)	Purpose of Files within this Directory
usbkern/	This is the top-level directory for all DataPump-related files.
usbkern/app	Each sub-directory of this directory contains all of the source, build, and USB resource files for a specific high level application to run on the Target Hardware. See *Section 2.2.4* Demo Applications, for an overview of each of the supported applications. These applications use the protocol and hardware libraries to accomplish their tasks and may be modified to tailor to a more specific application.
usbkern/arch	Each sub-directory of this directory contains all of the files required to build object code for the support of a specific Target CPU. For each CPU, it contains (1) boot code specific to the processor and Target Operating System, (2) some hardware level interface code specific to the architecture (e.g. interrupt vector interface code, and (3) m4 script files for conversion translation support of the specific CPU's assembly language.
usbkern/bin	INTERNAL USE ONLY Contains 3 shell script files (.sh) potentially for internal use in the building of zip files and other partial distribution builds.
usbkern/build	When using the MCCI build system, this is the default target directory of the final build files as well as all intermediate link and compilation objects. When the "make build tree" process is complete, there will be a specific set of sub-directories created within this directory for the specific implementation specified.
usbkern/common	This directory contains all of the core/common USB code.
usbkern/doc	
usbkern/i	This directory contains all of the common include files used by the DataPump software contained in the usbkern/common directory.
usbkern/host	This directory contains all source files for the host stack.

Directory Name (under MCCI/ directory)	Purpose of Files within this Directory
usbkern/libport	This directory contains the all of the files required for a specific compiler to be used with a specific hardware architecture. It also includes some documentation on the process to port to a new compiler and/or hardware platform. See the *MCCI USB DataPump Porting Reference Manual* for more details.
usbkern/m4	The DataPump software utilizes a macro processor tool called *bsdm4* for processing source code files written in a language called *m4*. The DataPump software uses the *m4* language to write assembly code that is independent of processor type. The output of the *bsdm4* tool is the assembly language for a specific Target CPU. This approach is only used for hardware level interface code that is difficult or impossible to do at the C language level in a platform independent manner.
usbkern/mk	This directory contains the core makefiles to be used with *bsdmake* build utility for all types of builds. See “readme” file in this directory for more specific information on the files contained in this directory.
usbkern/os	The DataPump software has an operating system abstraction layer to provide operating system type function calls independent of the operating system actually used. This directory contains all of the source files required to interface this independent software interface to a specific Target Operating System (e.g. none, pSOS, ThreadX, PowerTV).
usbkern/proto	This directory contains all of the source and build files for the protocols supported across the USB. A particular application should require only one of these USB protocols or a custom protocol to be developed. Each sub-directory of this directory contains all of the source and build files for a specific protocol to run on the Target Hardware. See *Section 2.2.3 Device* Class Protocol for an overview of each of the protocols. These protocols may use built in or additional hardware libraries to accomplish their tasks depending on the particular protocol.

Appendix C DataPump File Types

The following table is provided as a quick reference to help the user learn the purpose of files within the DataPump directory structure.

Table C-1. DataPump File Extensions

File Extension	File Type
.a	Library (“archive”) file containing compiled DataPump code; used primarily for embedded system targets.
.asm	Assembly language source file (when using Microsoft compilers)
.bat	Batch file for execution on Microsoft operating systems only.
.c	C language file
.d	C dependency file, automatically generated by the MCCI build process.
.lib	library file containing compiled DataPump code; used primarily when compiling code with Microsoft tools.
.h	Standard C language “include” file, used to hold common definitions of macros, data structures, etc.
.inc	A text file to be included in another. Primarily used within .mk files. Helps to break up the build process into orthogonal or manageable chunks.
.m4	An m4 language file. m4 is a macro language used by the DataPump to create assembly lanuage files independent of a specific processor assembly language. MCCI provides bsdm4 for processing these files.
.mk	Makefile for use with the bsdmake build utility
.o	Object file
.obj	Object file (primarily used when compiling with Microsoft tools)
.pdf	A standard document format which can be read by Adobe Acrobat product.
.pkl	A packlist file.
.s	Assembly language source file (when using non-Microsoft compilers)
.sh	A TTK (Thompson Toolkit) Unix-like shell file (similar to a batch file).
.sm4	An assembly-language file that should be pre-processed with m4 before assembly.

File Extension	File Type
.txt	A text file that can be read by any text editor.
.urc	USB Resource Compiler output file used by the MCCI USB Resource Compiler to create USB device resource descriptors.
.var	A text file containing definitions to scanvars, an MCCI-supplied tool that collects settings from a set of files.
.zip	A compressed output file of the pkzip file compression application.
No extension	Usually either a text file or an application file, depending on the compiler in use.