Application Note

AN_402

MCCI USB DataPump User Guide

Version 1.0

Issue Date: 2017-09-13

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

Bridgetek Pte Ltd (BRTChip)
178 Paya Lebar Road, #07-03, Singapore 409030
Tel: +65 6547 4827 Fax: +65 6841 6071
Web Site: http://www.brtchip.com

MCCI Legal and Copyright Information

Copyright:

Trademark Information:

The following are registered trademarks of MCCI Corporation:

- MCCI
- TrueCard
- TrueTask
- MCCI USB DataPump
- MCCI Catena

The following are trademarks of MCCI Corporation:

- MCCI Skimmer
- MCCI Wombat
- InstallRight
- MCCI ExpressDisk

All other trademarks, brands and names are the property of their respective owners.

Disclaimer of Warranty:

MCCI Corporation ("MCCI") provides this material as a service to its customers. The material may be used for informational purposes only.

MCCI assumes no responsibility for errors or omissions in the information contained at the world wide web site located at URL address 'http://www.mcci.com/', links reachable from this site, or other information stored on the servers 'www.mcci.com', 'forums.mcci.com', or 'news.mcci.com' (collectively referred to as "Web Site"). MCCI further does not warrant the accuracy or completeness of the information published in the Web Site. MCCI shall not be liable for any special, indirect, incidental, or consequential damages, including without limitation, lost revenues or lost profits, which may result from the use of these materials. MCCI may make changes to this Web Site, or to the products described therein, at any time without notice. MCCI makes no commitment to maintain or update the information at the Web Site.

THESE MATERIALS ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

Table of Contents

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- and 32-bit processors.

2 DataPump Product Overview

2.1 DataPump Base Product vs Application/Protocol Add-ons

2.1.1 Code

- Header Files
- Device Stack

2.1.2 Tools and Build System

- Building Tools
- BSDmake
- USBRC
- 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, and

C. destination port software.

For the protocol and application layers, MCCI offers several out of the box protocol and application layers and supports the creation of custom protocol and application layers. Since the destination port software is customer specific, most of this is left to the customer.

Figure 1 shows the overall software/hardware architecture of the DataPump's intended usage.

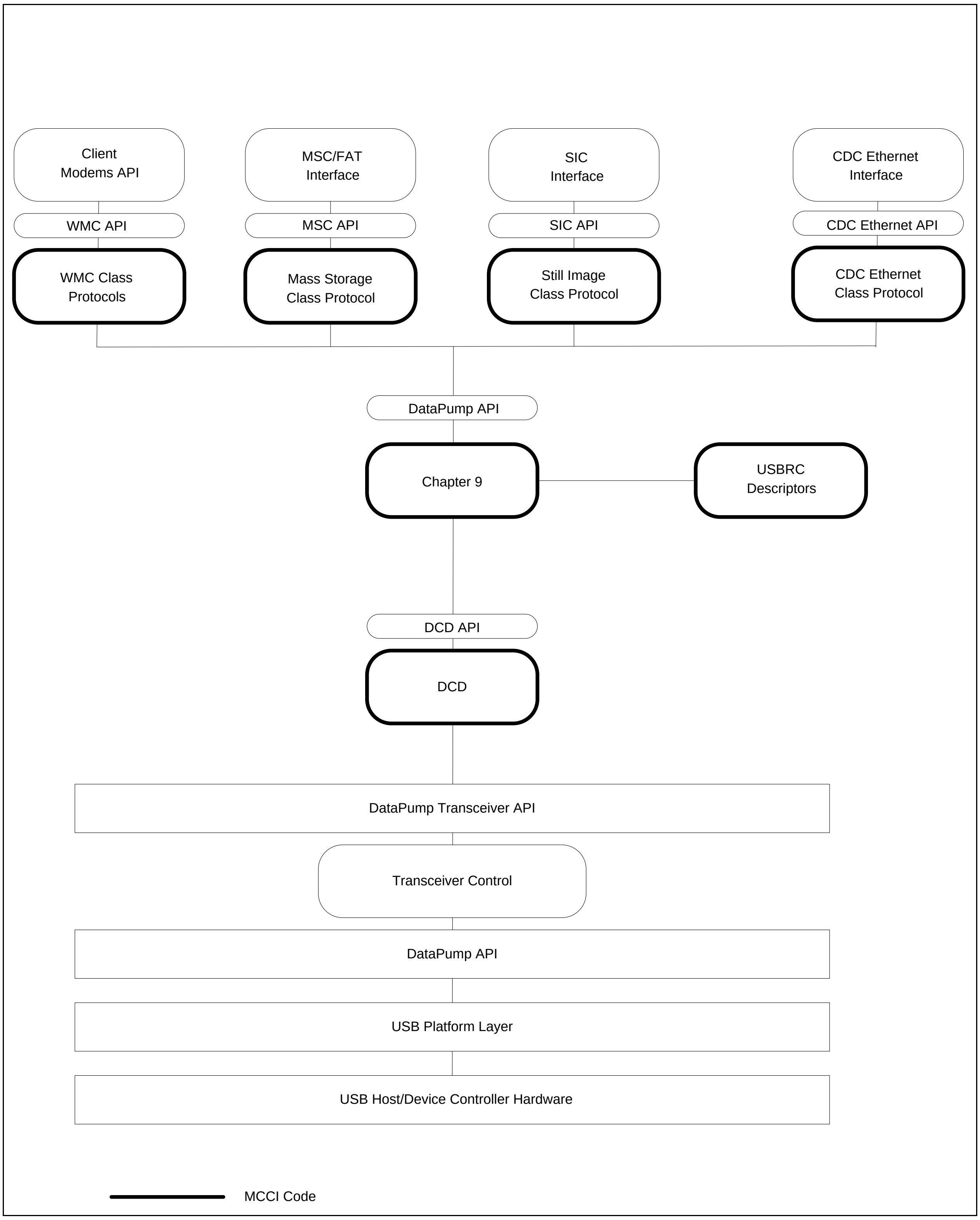

Figure 1 Architecture of Target Hardware/Firmware

2.1.3 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 allows upper-edge clients to treat them in a common way, if the system engineer finds this appropriate.

Protocol Modules generally operate by creating DataPump objects (see section 7 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. You modify your URC file to incorporate the interfaces, endpoints and descriptors appropriate to the device class.

2. If you're using the MCCI build system, you modify your application's UsbMakefile.inc build control file to reference the public header files and libraries for the protocol module.

3. You modify your 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 your run-time image.

4. You 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 give a list of the MCCI supported protocols that are either included with the base product or may be purchased separately directly from MCCI.

2.1.3.1 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.

Please refer to the AN_405 MCCI-USB-DataPump-HID-Protocol-Users-Guide for more information.

2.1.3.2 Networking Related Protocols (cdcether, vether, rndis)

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

The base protocol created is the CDC Ethernet protocol (*cdcether*). The DataPump CDC Ethernet protocol is a USB Communication Device Class compliant protocol. It receives standard USB CDC communication commands and data and transmits them to a higher-level protocol or application.

The Virtual Ethernet device (*vether*) protocol uses the CDC Ethernet protocol to create a virtual Ethernet device that echoes back whatever is sent to it. This is the networking equivalent of the loopback application for use as a sample, for debugging, and performance testing.

An optional final layer of protocol is the Remote NDIS protocol. NDIS or Network Driver Interface Specification is a Microsoft specification for standard networking interfaces. RNDIS is this specification applied to remote networking interfaces. The *rndis* protocol implements a Microsoft compliant RNDIS layer on top of the CDC Ethernet protocol layer.

Please refer to their corresponding protocol reference manuals for more information:

AN_406_MCCI-USB-DataPump-Virtual-Ethernet-Protocol-UserGuide

2.1.3.3 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 2.1.5.2 Mass Storage Class Demo (mscdemo) for a description of an application using this protocol.

Please refer to the AN_403 MCCI USB DataPump Mass Storage Protocol User Guide for more information.

2.1.4 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 on your own, refer to the roadmap in the previous section and Section 5 Implementing A Custom Protocol or Application.

2.1.4.1 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.

2.1.4.2 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.

2.1.4.3 Remote NDIS Bridge (rndisbrg)

NDIS or Network Driver Interface Specification is a Microsoft specification for standard networking devices. The *rndisbrg* or remote NDIS bridge application uses the MCCI CDC Ethernet Protocol and Virtual Ethernet protocol to create a standard Microsoft NDIS compliant device. All you need to do is configure and build the code and you have a fully executable USB networking device. You may edit the application to tailor it to any specific extra or different capabilities that you desire.

3 DataPump Development Overview

3.1 Overview

This section provides information relating to how to use the base DataPump product along with some background information.

3.1.1 DataPump Usage of Third Party Tools

In order to support a large combination of Target CPU, Target compiler, and Target Operating System 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 follows.

Third Party Software	Purpose
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.

Table 1 Build Tools used by the DataPump

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.
- 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.

Although the process has several steps, you do not need to become a USB expert in order to use the DataPump. Using the MCCI USB DataPump and the supplied loopback protocol, MCCI's customers have demonstrated functionality on their prototype boards within a matter of days, with no prior USB experience.

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.

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 Unicode, in the byte order used by Intel systems, which can make it a little awkward to prepare. However, MCCI provides a tool that makes it easy to prepare these strings, even if the target compiler doesn't support UNICODE in Intel byte order. The MCCI USBRC resource compiler is such a tool.

Figure 2 is a schematic diagram showing many of these features.

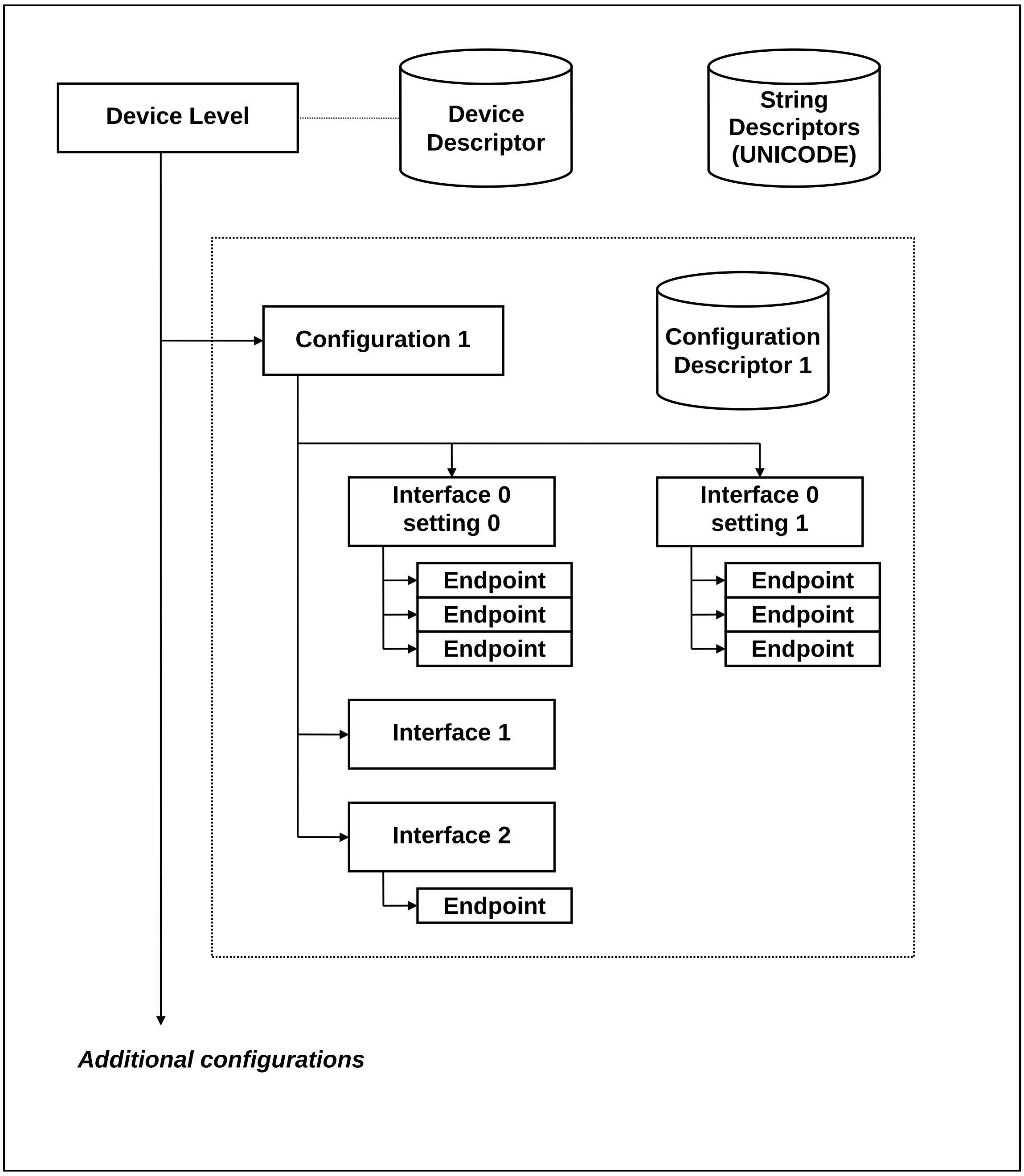

Figure 2 USB Device Architecture

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. In the USB 1.0 specification, Interrupt endpoints were only defined for device-to-host transfers. In the USB 1.1 specification, Interrupt endpoints have been made symmetrical, so there are also host-to-device interrupt 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 transfer rate. However Interrupt endpoints take much less bandwidth than Isochronous endpoints, at the cost of longer 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 AN_400_MCCI_USB_Resource_Compiler_UserGuide 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 Lucent USS820 USB controller declares an EPIO820 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 EPIO820 by the USS820-specific code. The same is true for the UDEVICE structure, the port declares a UDEV820 structure that has additional port specific information appended to the UDEVICE structure.

Figure 3 is a schematic of these data structures.

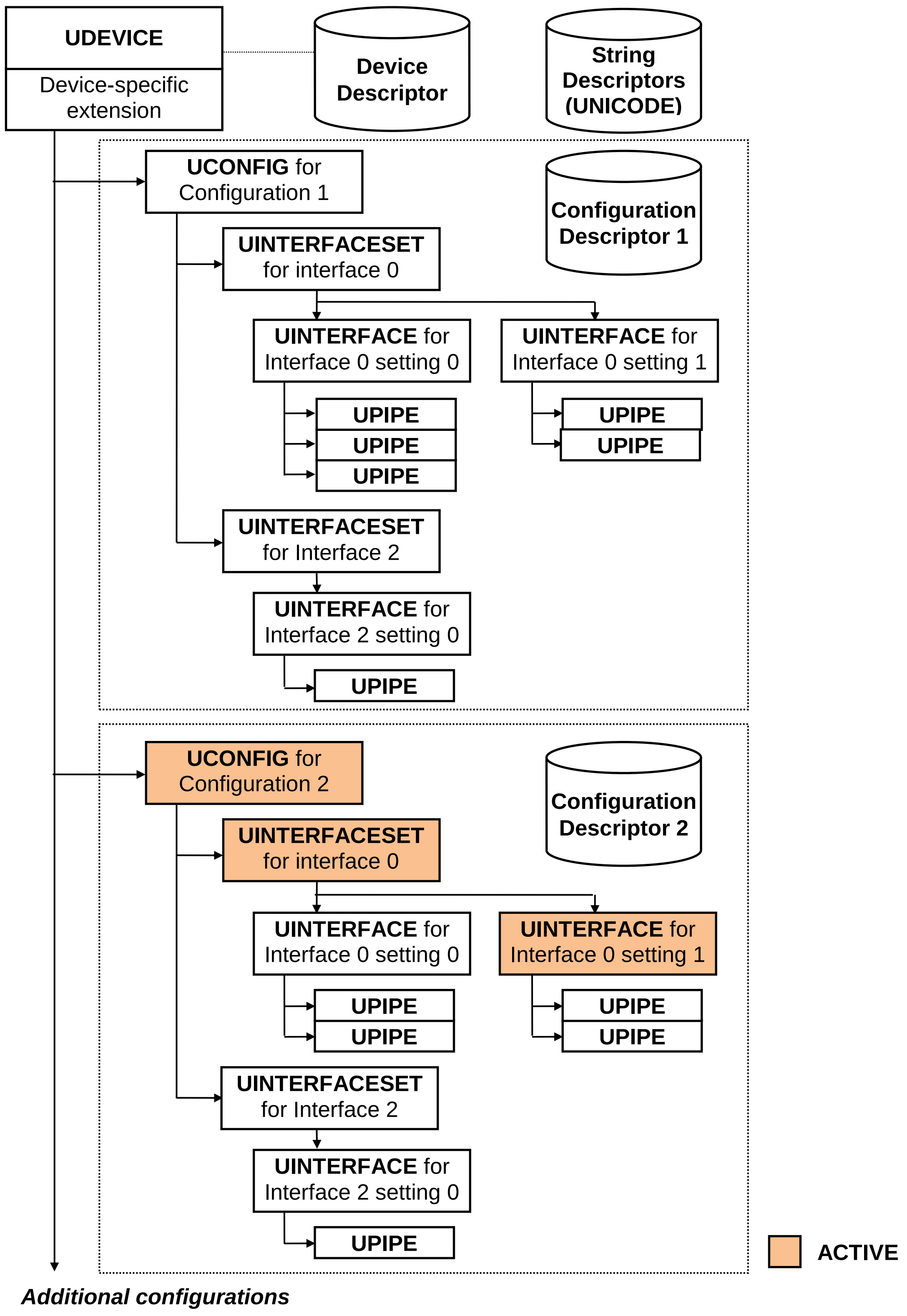
UDEVICE
Device-specific extension
Device Descriptor
String Descriptors (UNICODE)
UCONFIG for Configuration 1
Configuration Descriptor 1
UINTERFACESET for interface 0
UINTERFACE for Interface 0 setting 0
UINTERFACE for Interface 0 setting 1
UPIPE
UPIPE
UPIPE
UPIPE
UPIPE
UINTERFACESET for Interface 2
UINTERFACE for Interface 2 setting 0
UPIPE
UCONFIG for Configuration 2
Configuration Descriptor 2
UINTERFACESET for interface 0
UINTERFACE for Interface 0 setting 0
UINTERFACE for Interface 0 setting 1
UPIPE
UPIPE
UPIPE
UPIPE
UINTERFACESET for Interface 2
UINTERFACE for Interface 2 setting 0
UPIPE
ACTIVE
Additional configurations

Figure 3 SB 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.

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, you 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 4. 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 our 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.

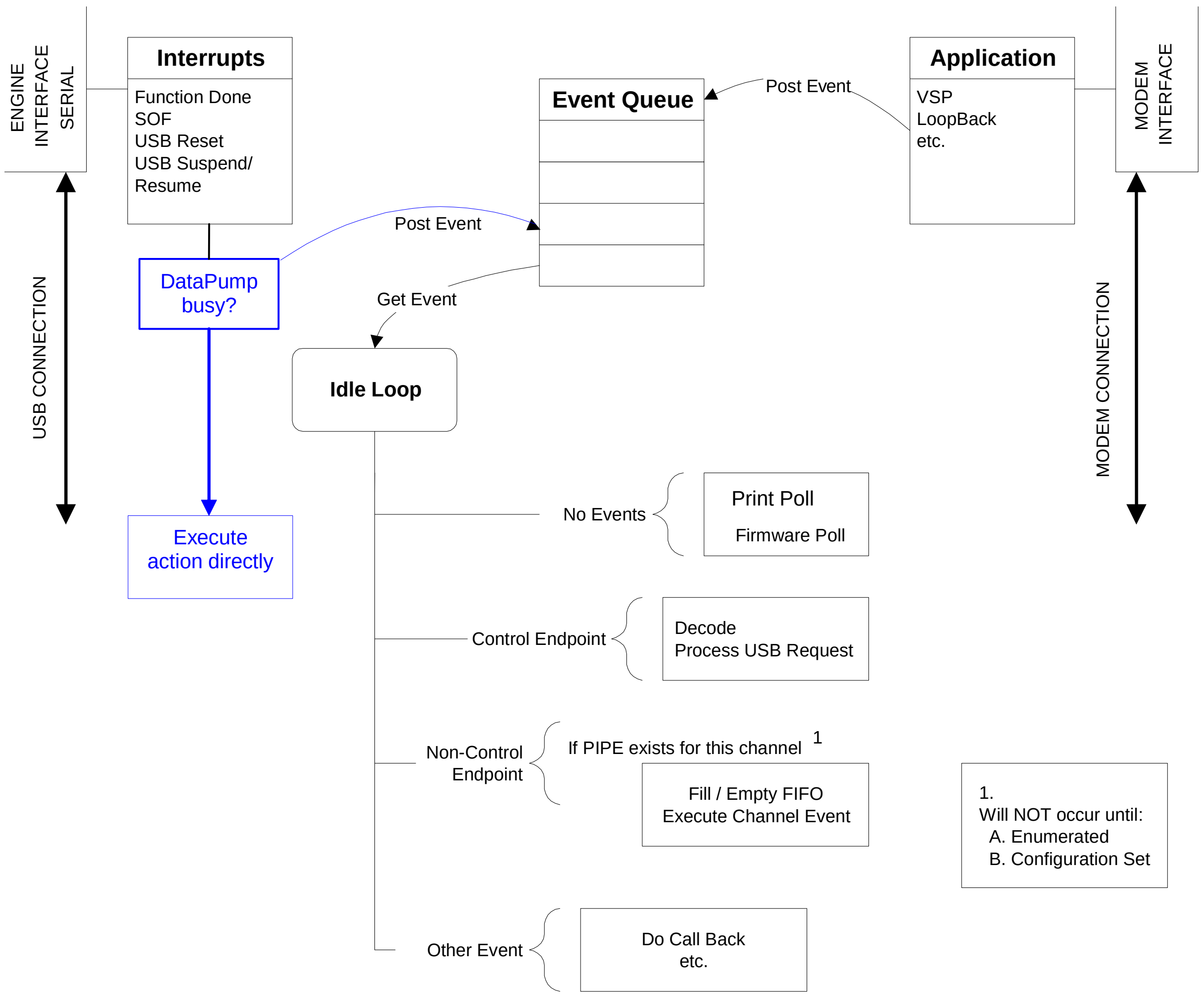

Figure 4 Event Queue Processing

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 5). 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 the 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.

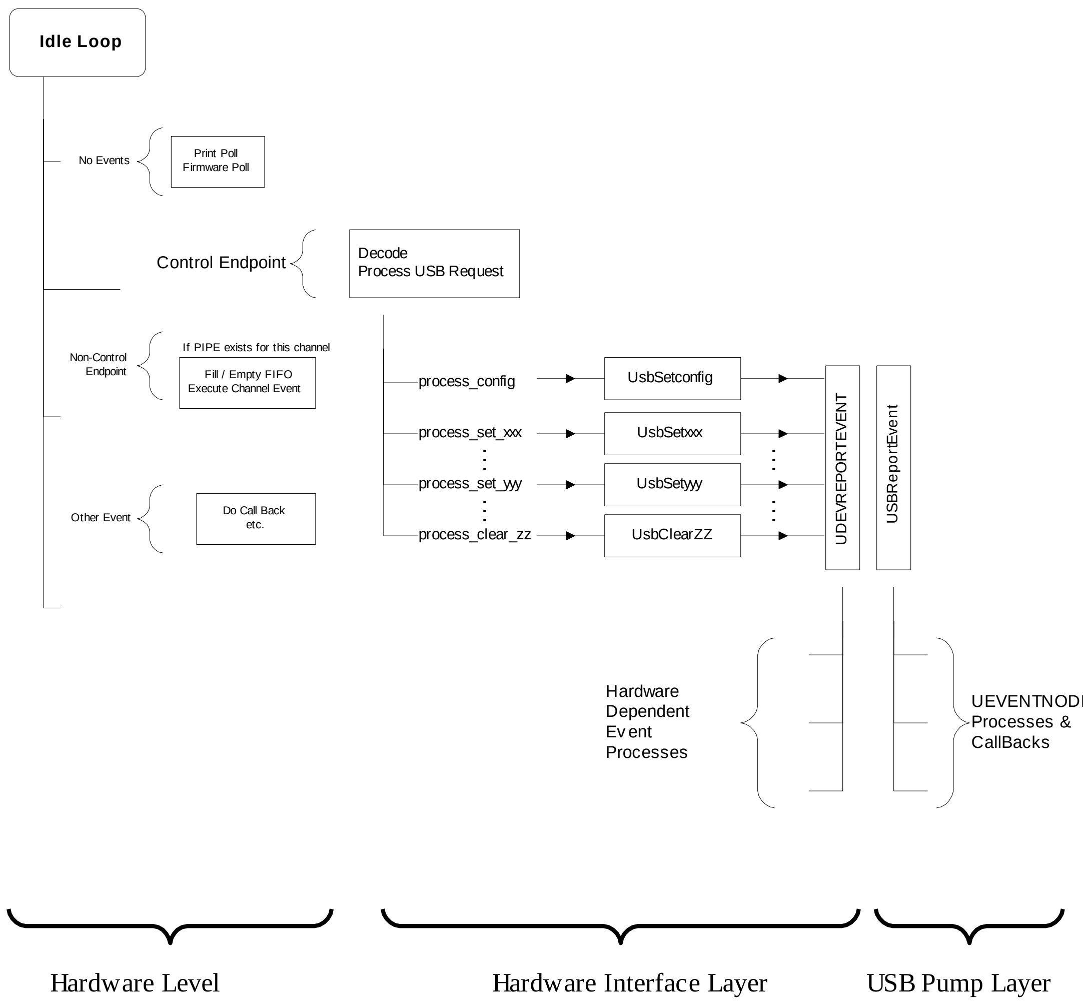

Figure 5 Control Endpoint Processing

5 Implementing a Custom Protocol or Application

Adding a custom protocol or application (or modifying an existing one) 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 protocol/application files).

2. Edit these source files.

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

The following two sections 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 (See Figure 6).

1. Begin by specifying how the device will appear "on the wire." You must specify the following information:
   - 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.

   At the end of this process, you will have all the information you need to verify that the silicon you want to use will do the job. You will also be able to create a prototype USB resource file that describes your device.

2. Next, you 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 for free from the MCCI web site, http://www.mcci.com.

3. Once you have described the peripheral, 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. Next, you must create a simple initialization function that attaches any protocols you need onto the USB interface.

   With these three pieces, 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, you write the application code that calls the "top edge" of the protocols according to the protocol APIs.

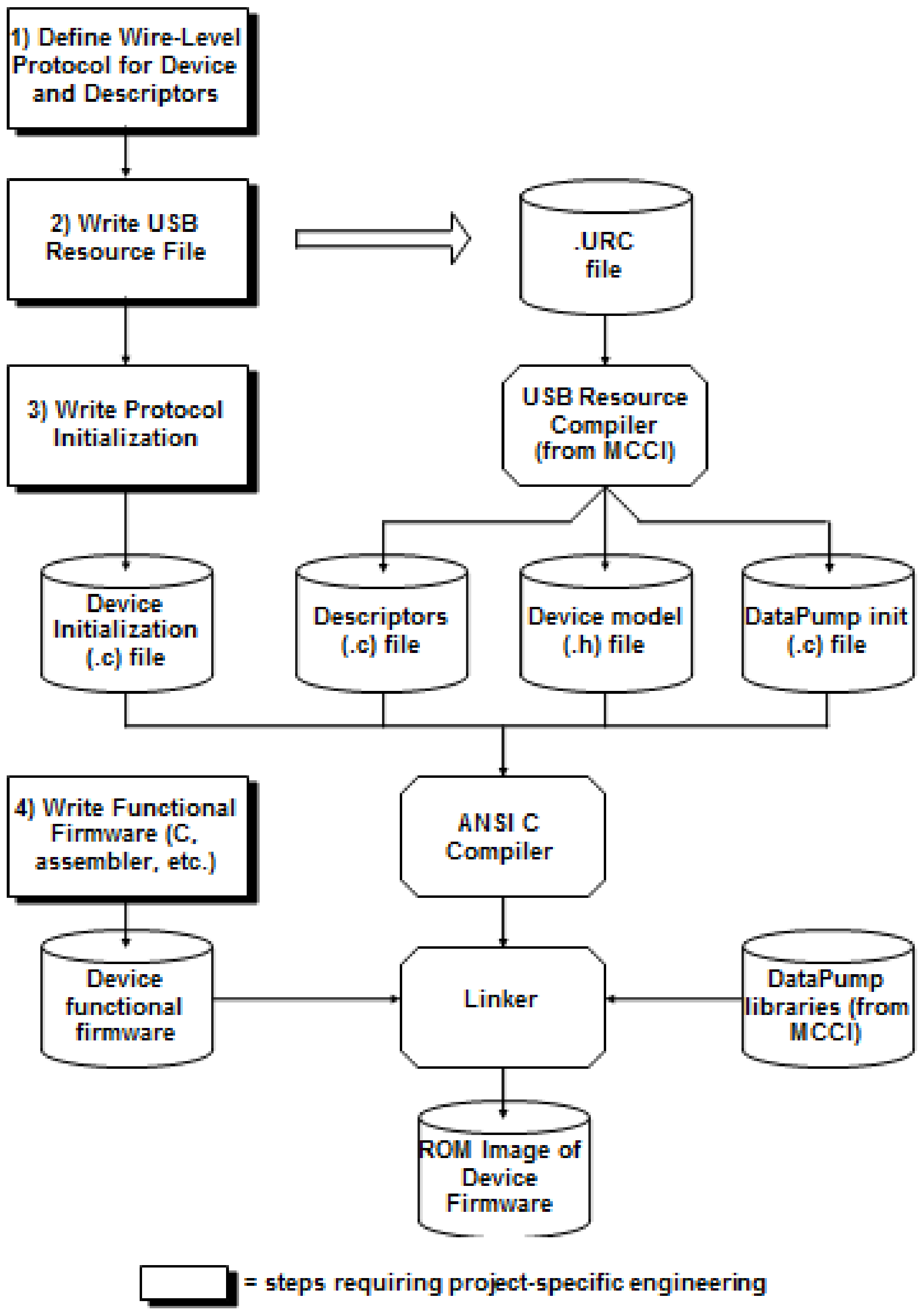
1) Define Wire-Level Protocol for Device and Descriptors
2) Write USB Resource File
.URC file
3) Write Protocol Initialization
USB Resource Compiler (from MCCI)
Device Initialization (.c) file
Descriptors (.c) file
Device model (.h) file
DataPump init (.c) file
4) Write Functional Firmware (C, assembler, etc.)
ANSI C Compiler
Device functional firmware
Linker
DataPump libraries (from MCCI)
ROM Image of Device Firmware
= steps requiring project-specific engineering

Figure 6 Design Flow

6 MCCI USB DataPump Data Structures

6.1 USB Device Representation

6.1.1 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 L PM 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. You can think of UDEVICE as representing this hardware. If you're writing reentrant client code, you can use UsbPumpObject_GetDevice() 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.

6.1.1.1 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("*")
    );
```

6.1.2 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.

6.1.2.1 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
       )
```

6.1.3 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

6.1.3.1 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
       )
```

6.1.4 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 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.

6.1.4.1 Uifc_bStatus – 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
       )
```

6.1.5 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.

6.1.6 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 6.1.7 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.

6.1.6.1 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 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
               )
```

6.1.6.2 Initializing the Endpoint

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

6.2 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.)

6.2.1 UEVENT

The UEVENT is the base type for holding event codes.

6.2.2 UEVENTFN

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
               );
```

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
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.

Symbolic Name	Value	evinfo->	Description
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)
UEVENT_PLATFORM_EXTENSION	19	UEVENTPLATFORMEXTENSION_INFO	a platform-specific extension -- this is for use by platform-

Symbolic Name	Value	evinfo->	Description
			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_ISOC H_DELAY_INFO	post to DCD to set isochronous delay.
UEVENT_FUNCTION_SUSPEN D	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_C APABLE_INFO	post to interface to get function remote wake capable.
UEVENT_DEVICE_NOTIFICAT ION	31	UEVENT_DEVICE_N OTIFICATION_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 chip driver to exit U1 or U2 sleep state.

Table 2 Defined UEVENT Codes

6.2.3 UEVENTNODE

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

uev_next	A forward link pointer to the next event node in the queue.
uev_last	A backward link pointer to the previous event node.
uev_pfn	The callback function for this event node.
uev_ctx	is the context pointer for use by the callback function.

Here is an example of using the events structure and method to retrieve a configuration value selected by the 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 */
   }
}
```

6.2.4 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	The feature selector. This corresponds to the wValue element of a device request packet.
uef_index	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	The raw setup packet.
uef_relstruct	The 'relevant' structure. This union contains a pointer type for each of the base types used to represent a USB device.

6.2.5 USETUP

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.

6.2.6 UEVENTSETUP

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 either set the accept field to accept, or set the reject field to reject; or else leave both fields alone, so that 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	the unpacked version of the setup packet data.

6.3 Platform

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

6.3.1 UPLATFORM Type Derivation Diagram

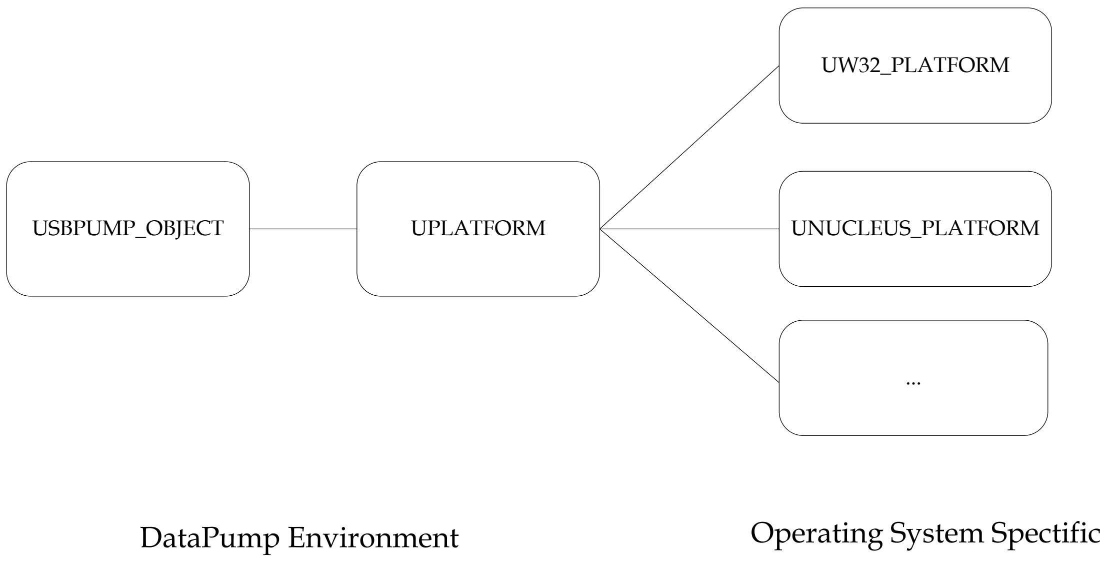

Figure 7 Platform Type Derivation

6.3.2 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
These 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 interfaces provide some additional control in non-preemptive systems

upf_pDebugPrintControl

This 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.

6.3.3 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 out switch. 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.

6.3.4 UDATASTREAM

```
#include "udatastream.h"

struct __TMS_UDATASTREAM
   {
   UDATASTREAM  *pNext;
   UDATASTREAM  *pLast;
   UPIPE        *pCurrentPipe;
   UBUFQE       *pHoldQueue;
   UDATAPLANE   *pDataPlane;
   UCHAR        ucBindingFlags;
   UCHAR        ucPipeOrdinal;
   USHORT       usEpAddrMask;
   };
```

6.4 HIL Structures

6.4.1 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.

7 MCCI DataPump Object System

7.1 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.

7.2 Properties of Objects

DataPump objects are used to collect and represent all the major data structures within the DataPump. They have the following common properties:

7.2.1 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.

7.2.2 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, you 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.

7.2.3 Objects Have Behavior

In the MCCI 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 it 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).

7.2.4 Objects Have Relationship 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.

7.3 USBPUMP_OBJECT_HEADER

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

Definition: USBPUMP_OBJECT_HEADER has the following contents:

```
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;
```

Description: 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, and is 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.

7.4 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 associated with it an IOCTL function provided by a class-specific function. 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 want to have 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.

Returns: Any USBPUMP_OBJECT_IOCTL_FN must return either 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).

7.5 USBPUMP_OBJECT_LIST

Function: Provide a standard doubly-linked list component for

USBPUMP_OBJECT_HEADERs.

Definition: USBPUMP_OBJECT_LIST has the following contents:

USBPUMP_OBJECT_HEADER *pNext;

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, we define a structure that just contains the pNext and pLast for the particular list.

7.6 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. So for example, we have:

```
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. So for example, we require that if `OBJECT1`, `OBJECT2` and `OBJECT3` are all derived from `OBJECT`, that the common fields in `OBJECT1`, `OBJECT2` and `OBJECT3` that are 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. We assume that you'll create a structure and a union. The union is the top-level object, and allows your object to be viewed either in its concrete form or in its abstract form. The structure gives the concrete 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, we define two macros, 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 we have `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`. We can then define the derived type `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`, you still need cast, e.g., `((OBJECT2 *)&(pObject)->ObjectCast`.

There is a new way 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's always better to use a field "Tag"; that way 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, we can define a macro like:

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

7.7 MCCI Objects Hierarchy

Device Stack: Root object is the *default* IOCTL parent of every other object.

Every UPLATFORM is an object; it's 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 we use this. Customers want to change the behavior of the DataPump based on MMI settings (mass storage only or modem only). So the platform must provide an implementation of USBPUMP_IOCTL_GET_VIDPIDMODE in the UPLATFORM-outcalls for your platform. Normally, there's a place in the OS-specific init code (e.g. the args to unucleus_UsbPumpInit ()) where a client can pass a pointer to an IOCTL function. As soon as this behavior is implemented, DataPump will automatically start tracking the MMI setting.

7.8 MCCI Objects Functions

7.8.1 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.

7.8.2 UsbPumpObject_Init

Function: Fill in an object header, and register 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.

7.8.3 UsbPumpObject_DeInit

Function: Unregisters an object.

Definition:

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

Description: The object is de-registered. We check for a multiple de-init: we clear the link fields after de-registering. A dual deregister causes us to issue a software-check. We're careful to leave enough linkage in place to allow us to find the UPLATFORM, in case we need to display a message.

Returns: No explicit result.

7.8.4 UsbPumpObject_EnumerateMatchingNames

Function: Scan 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.

7.8.5 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.

7.8.6 UsbPumpObject_FunctionClose

Function: Close 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.

7.8.7 UsbPumpObject_GetDevice

Function: Given some object, find 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.

7.8.8 UsbPumpObject_GetRoot

Function: Given some object, 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.

7.8.9 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), and 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.

7.8.10 UsbPumpObject_SetDebugFlags

Function: Set debug flags for given object.

Definition:

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

);
```

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

Returns: No explicit result.

7.8.11 UsbPumpObject_GetDebugFlags

Function: Get debug flags for given object.

Definition:

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

Description: This routine returns debug flags for given pObjectHeader.

Returns: Debug flags.

8 MCCI Event Handling

8.1 Event Support Function

8.1.1 UsbPostIfNotBusy

Function: Post a callback completion unless it's 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.

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

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 for us to guard the update.

8.1.2 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. This 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.

8.1.3 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. This can be used to compress completion events into a single dispatch of the completion function.

Returns: None.

9 MCCI Dynamic Memory Allocation Routines

9.1 Memory Functions in Pre-2.0 DataPump

9.1.1 UsbAllocateDeviceBuffer

Function: Allocate 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, we implement the concept of device pool which is intended 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 implementer 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.

9.1.2 Memory Allocation API Changes

9.1.2.1 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, we have 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 in order to take advantage of future enhancements in the `UPLATFORM` api.

9.1.2.2 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, we define two types:

The abstract pool object itself is a union, containing a single view:

```
USBPUMP_ABSTRACT_POOL_CONTENTS  AbstractPool;
```

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, you may 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.

10 MCCI USB DataPump Internal API

10.1 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. In the MCCI DataPump, we use 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 that 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 that the system engineer be careful about what's 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.

10.1.1 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 provides the vector of application initialization nodes and a pointer to a function to be called after initialization is complete. This function is specific to your needs. One UDEVICE is created for each app init node element that "matches".

10.1.2 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
       );
```

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. Pre-function and post functions are hooks for doing special things, normally not 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"

Here is 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, so we don't care about dev class etc.
- USB_bInterfaceClass_MassStorage constants come 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's 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 explain how it's used.

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's mass storage. If so, a new mass storage instance is created for that interface, and that interface is marked "in use". The framework then 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".

10.1.3 Port Init Header

Port is for USB chip hardware. Port initialization is like 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 chip kern head 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 chip and wireless USB chip, 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).

10.2 Device Related Functions

10.2.1 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

Returns: Pointer to the data buffer, or NULL.

10.2.2 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

11 MCCI USB DataPump API

11.1 Debugging Functions

11.1.1 UsbDebugLogf

Function: printf() version for logging; it's 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)
```

Note: This function has been replaced by UsbPumpDebug_PlatformLogf(), and is provided only for backwards compatibility.

11.1.2 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.

Mask Name	Value	Description
UDMASK_FATAL_ERROR	0	pseudo level: fatal error
UDMASK_ERRORS	1L << 0	trace gross errors

Mask Name	Value	Description
UDMASK_ANY	1L << 1	catch-all category
UDMASK_FLOW	1L << 2	flow through the system
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

Table 3 Description of debug mask

11.2 Timer API

When polling or otherwise operating finite state machines, there's a need for periodic events driven from a reference source. This timing may be needed even if a given HCD is not operating, and so rather than depend on HCD capabilities we provide one 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 post condition, 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:

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)

Field	Description
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's an implementation decision)

Table 4 USBPUMP_TIMER Contents

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 simply removing the object from the queue it's 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.

11.2.1 Timer Implementation Framework

Timer support is added to the platform by adding two fields to the UPLATFORM, described in Table 5.

Field	Description
CONST USBPUMP_TIMER_SWITCH *pPlatform->upf_pTimerSwitch;	Pointer to table of dispatch functions.
VOID *pPlatform->upf_pTimerContext;	Context pointer for timer implementation

Table 5 UPLATFORM additions for timer support

The USBPUMP_TIMER_SWITCH contents are described in Table 6.

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

Field	Description
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

Table 6 USBPUMP_TIMER_SWITCH Contents

The types are described in subsequent sections.

The timer context pointer is provided for two reasons:

1. It allows us to replace the DataPump standard timer implementation with a platform specific one, in case the DataPump's implementation is not suitable.
2. It allows us to omit timer support when it is not required. For example, device-only applications of the DataPump typically do not need timer support.

11.2.1.1 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
          );
```

11.2.1.2 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
          );
```

11.2.1.3 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
          );
```

11.2.1.4 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.)

11.3 Miscellaneous Functions

All of the following functions are defined in the USB DataPump header file, "usbpump.h".

11.3.1 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 aren't sure if the buffer is addressable. We copy the data to a supplied bounce-buffer, and then send it on to the host.

Returns: Actual number of bytes transferred.

11.3.2 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: It is important that ReplyMax be identical to the original wLength of the setup packet; otherwise, mysterious data transfer hang-ups may result on chips like the USS820, D12, or BCM3310.

11.3.3 UsbPumpLib_BufferCompareString

Function: Compare buffer to string for equality.

Definition:

```
BOOL
UsbPumpLib_BufferCompareString(
CONST TEXT *buf,
BYTES n,
CONST TEXT *s
);
```

Description: Each character in buf must match the corresponding character in s, or we will declare no match. lenstr(s) must be equal to n, or we will declare no match.

Returns: TRUE if match, FALSE otherwise.

Returns TRUE if the pointers are equal, otherwise returns FALSE if either pointer is NULL.

11.3.4 UsbPumpLib_BufferFeildlIndex

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.

We'll also return (n) in case an error occurs, e.g. (i > n) or (buf EQ 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).

11.3.5 UsbPumpLib_BufferFeildlLength

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. We simply scan forward from the specified position until we find the next delimiter, or until we get to the end of the buffer. 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, we skip any leading whitespace, then skip to the end of the next field; we return the number of characters skipped.

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.

11.3.6 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.

11.3.7 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 comparison, 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.

11.3.8 UsbPumpLib_SafeCopyBuffer

Function: 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)[1]. 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.

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).

11.3.9 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).

[1] [0, n) denotes the range from 0 to n-1

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 you should assume that one or more bytes of the string was truncated. If nBuffer>0, and iBuffer + the result == nBuffer-1, then the string may have been truncated.

11.3.10 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.

11.3.11 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.

11.3.12 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 we suppress the leading '+' we'd ordinarily put if the number is positive.

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), we won't do anything; we'll always return 0.

If (n == 1), all that happens is that a '\0' is placed into buf[0].

11.3.13 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's no way to limit the number of significant digits independent of the buffer size.

11.3.14 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.

11.3.15 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.

11.3.16 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.

11.3.17 UsbPumpLib_Best1ToMicroSecond

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. We convert the encoded value to micro seconds.

Returns: No explicit result.

11.3.18 UsbPumpLib_SHA1_Init

Function: Initialize the SHA1 context.

Definition:

```
USTAT UsbPumpLib_SHA1_Init (
USBPUMP_SHA1_CONTEXT *    pSha1Ctx,
USBPUMP_OBJECT_HEADER *   pObjectHeader
);
```

Description: Function is used for initializing the hash value for SHA-1.

Returns: USTAT_OK if success, otherwise USTAT error code.

11.3.19 UsbPumpLib_SHA1_Update

Function: Hash computation

Definition:

```
USTAT UsbPumpLib_SHA1_Update (
CONST USBPUMP_SHA1_CONTEXT *    pSha1Ctx,
CONST UCHAR *           pMsgBuff,
unsigned long           MsgLength
);
```

Description: It compute cumulative message length, intermediate hash value, Residual message length, which is equal to number of message bits in the last unprocessed block and convert these message bits into word (32 bits) value. Computed values are updated in SHA1Ctx structure which is used in next UsbPumpLib_SHA1_Update or in UsbPumpLib_SHA1_Final.

Returns: USTAT_OK if success, otherwise USTAT error code.

11.3.20 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.

11.3.21 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, you may want to generate a different sequence each time you use the generator. In that case, you must get non-zero values for at least some of the init_x, init_y, init_z and init_c. A common way to do this is to use the time of day. If your system provides good sources of entropy-based random numbers, you can also use that source to get 1 to 4 random values.

Returns: No explicit result. The contents of *p are updated in place.

11.3.22 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, obviously 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 you initialize it differently from run-to-run, you'll get the same sequence every time you use it. 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.

11.4 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’s considered to be base 16; otherwise if the number begins with `'0'`, it’s considered to be base 8; otherwise 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 7.

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

Table 7 Results Matrix for UsbPumpLib_BufferTo... Routines

12 Basic HIL Functions

12.1 Application Hooks

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".

12.1.1 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

12.2 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.

12.2.1 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 we don't assume alignment is OK.

Returns: The value.

Macro for invoking:

```
UHIL_LE_GETUINT16 (buf)
```

12.2.2 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 we don't assume alignment is OK.

Returns: The value.

Macro for invoking:

```
UHIL_LE_GETUINT32 (buf)
```

12.2.3 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 we don't assume alignment is OK.

Returns: The value.

Macro for invoking:

```
UHIL_LE_GETUINT64(buf)
```

12.2.4 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 we don't assume alignment is OK.

Returns: No explicit result; *pResult is updated in place.

Macro for invoking:

```
UHIL_LE_GETUINT128_S(pResult, pBuffer)
```

12.2.5 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, we break the value up into two 64-bit values.

Returns: No explicit result; *pResult is updated in place.

Macro for invoking:

```
UHIL_LE_GETINT128_S(pResult, pBuffer)
```

12.2.6 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: We simply crack the value into 2 bytes, and stuff it.

Returns: Nothing.

Macro for invoking:

```
UHIL_LE_PUTUINT16 (buf, val)
```

12.2.7 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: We simply crack the value into 4 bytes, and stuff it.

Returns: Nothing.

Macro for invoking:

```
UHIL_LE_PUTUINT32 (buf, val)
```

12.2.8 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 we don't assume alignment is OK.

Returns: The value.

Macro for invoking:

```
UHIL_BE_GETUINT16(buf)
```

12.2.9 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 we don't assume alignment is OK.

Returns: The value.

Macro for invoking:

```
UHIL_BE_GETUINT32(buf)
```

12.2.10 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 we don't assume alignment is OK.

Returns: The value.

Macro for invoking:

```
UHIL_BE_GETUINT64(buf)
```

12.2.11 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, we break the value up into two 64-bit values.

Returns: No explicit result; *pResult is updated in place.

Macro for invoking:

```
UHIL_BE_GETINT128_S(pResult, pBuffer)
```

12.2.12 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, we break the value up into two 64-bit values.

Returns: No explicit result; *pResult is updated in place.

Macro for invoking:

```
UHIL_BE_GETUINT128_S(pResult, pBuffer)
```

12.2.13 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: We simply crack the value into 2 bytes, and stuff it.

Returns: Nothing.

Macro for invoking:

```
UHIL_BE_PUTUINT16(buf, val)
```

12.2.14 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: We simply crack the value into 4 bytes, and stuff it.

Returns: Nothing.

Macro for invoking:

```
UHIL_BE_PUTUINT32(buf, val)
```

12.2.15 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: We simply crack the value into 8 bytes, and stuff it, from most significant to least.

Returns: Nothing.

Macro for invoking:

```
UHIL_BE_PUTUINT64(buf, val)
```

12.2.16 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, we break the value into two 64-bit values first.

Returns: No explicit result.

Macro for invoking:

```
UHIL_BE_PUTINT128_S(pBuf, pValue)
```

12.2.17 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, we break the value into two 64-bit values first.

Returns: Nothing.

Macro for invoking:

UHIL_BE_PUTUINT128_S(pBuf, pValue)

12.2.18 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);

12.2.19 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);

12.3 Library Functions

The following functions are provided for convenience.

12.3.1 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`.

12.3.2 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) Its 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.

12.3.3 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.

12.3.4 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.

12.3.5 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.

12.3.6 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.)

12.4 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.

12.4.1 UHIL_DebugPrintEnable

Function: Enable or disable debug printing.

Definition:

```
BOOL UHIL_DebugPrintEnable (
UPLATFORM *pPlatform,
BOOL enable
);
```

Description: We set the printing enable state to what the caller asks for and return the previous setting.

Returns: The previous enable state.

13 Contact Information

Headquarters – Singapore

Bridgetek Pte Ltd
178 Paya Lebar Road, #07-03
Singapore 409030
Tel: +65 6547 4827
Fax: +65 6841 6071

E-mail (Sales)	sales.apac@brtchip.com
E-mail (Support)	support.apac@brtchip.com

Branch Office – Taipei, Taiwan

Bridgetek Pte Ltd, Taiwan Branch
2 Floor, No. 516, Sec. 1, Nei Hu Road, Nei Hu District
Taipei 114
Taiwan, R.O.C.
Tel: +886 (2) 8797 5691
Fax: +886 (2) 8751 9737

E-mail (Sales)	sales.apac@brtchip.com
E-mail (Support)	support.apac@brtchip.com

Branch Office - Glasgow, United Kingdom

Bridgetek Pte. Ltd.
Unit 1, 2 Seaward Place, Centurion Business Park
Glasgow G41 1HH
United Kingdom
Tel: +44 (0) 141 429 2777
Fax: +44 (0) 141 429 2758

E-mail (Sales)	sales.emea@brtichip.com
E-mail (Support)	support.emea@brtchip.com

Branch Office – Vietnam

Bridgetek VietNam Company Limited
Lutaco Tower Building, 5th Floor, 173A Nguyen Van Troi,
Ward 11, Phu Nhuan District,
Ho Chi Minh City, Vietnam
Tel : 08 38453222
Fax : 08 38455222

E-mail (Sales)	sales.apac@brtchip.com
E-mail (Support)	support.apac@brtchip.com

Web Site

http://brtchip.com/

Distributor and Sales Representatives

Please visit the Sales Network page of the Bridgetek Web site for the contact details of our distributor(s) and sales representative(s) in your country.

Appendix A – DataPump Directory Structure

Since many types of applications, processors, chips, 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).

Directory Name (under MCCI/ directory)	**Purpose of Files within this Directory**
datapump	This directory is the main directory for the DataPump software. It contains the entire source and builds files for the hardware, application, and protocol specific aspects of the DataPump. It does not contain tools, documentation, or Wombat (MCCI WCDMA USB Test Board) specific files. The following few table entries describe the main sub-directories of this main directory.
datapump/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 2.1.5 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.
datapump/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.
datapump/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.
datapump/usbkern/build	This is the 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.
datapump/usbkern/common	This directory contains all of the core/common USB code.
datapump/usbkern/doc	
datapump/usbkern/i	This directory contains all of the common include files used by the DataPump software contained in the

Directory Name (under MCCI/ directory)	**Purpose of Files within this Directory**
	datapump/usbkern/common directory.
datapump/usbkern/libport	This directory contains the all of the files required for a specific compiler to be used with specific hardware architecture. It also includes some documentation on the process to port to a new compiler and/or hardware platform.
datapump/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.
datapump/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 the entire source files required to interface this independent software interface to a specific Target Operating System (e.g. none, pSOS, ThreadX, PowerTV).
datapump/usbkern/proto	This directory contains the entire source and builds 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 the entire source and builds files for a specific protocol to run on the Target Hardware. See Section 2.1.3 Protocol Modules 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 B – 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.

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 language 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

File Extension	File Type
	assembly.
.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, a 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.

Appendix C – Sample Hardware Interface Code

The following is C language code to be used as a shell/reference when implementing a new, non-USB hardware interface code.

```
UHIL_INTERRUPT_RESOURCE_HANDLE CONST     G_hRxInt         =
S3C4510_MAKE_INTERRUPT_RESOURCE_HANDLE(PORT_MPXINT_MACRX,FALSE);

UHIL_INTERRUPT_RESOURCE_HANDLE CONST     G_hTxInt         =
S3C4510_MAKE_INTERRUPT_RESOURCE_HANDLE(PORT_MPXINT_MACTX,FALSE);

UHIL_INTERRUPT_RESOURCE_HANDLE CONST     G_hIIC2Int =
S3C4510_MAKE_INTERRUPT_RESOURCE_HANDLE(PORT_MPXINT_IIC ,FALSE);

UHIL_INTERRUPT_RESOURCE_HANDLE CONST     G_Lxt971Int =
S3C4510_MAKE_INTERRUPT_RESOURCE_HANDLE(PORT_MPXINT_EXT2 ,FALSE);

UHIL_INTERRUPT_RESOURCE_HANDLE G_hBDMARxInt =
S3C4510_MAKE_INTERRUPT_RESOURCE_HANDLE(PORT_MPXINT_BDMARX,FALSE);

UHIL_INTERRUPT_RESOURCE_HANDLE G_hBDMATxInt =
S3C4510_MAKE_INTERRUPT_RESOURCE_HANDLE(PORT_MPXINT_BDMATX,FALSE);

UHIL_INTERRUPT_CONNECTION_HANDLE         G_hTxIntConn;

UHIL_INTERRUPT_CONNECTION_HANDLE         G_hRxIntConn;

UHIL_INTERRUPT_CONNECTION_HANDLE         G_hBDMARxIntConn;

UHIL_INTERRUPT_CONNECTION_HANDLE         G_hBDMATxIntConn;

UHIL_INTERRUPT_CONNECTION_HANDLE         G_hIIC2IntConn;

UHIL_INTERRUPT_CONNECTION_HANDLE         G_Lxt971IntConn;

/*

Name:    S3C4510_AdapterAttach

Function:

        Attach the adapter to the application.

Definition:

        BOOL

        S3C4510_AdapterAttach(

                PNIC_APPLICATION_CONTEXT self

                );

Description:

We initialize the adapter context and attach it and the adapter switch to the application context.
Then initialize the MAC registers themselves.

Returns:

        TRUE always.

*/

BOOL S3C4510_AdapterAttach(

        PNIC_APPLICATION_CONTEXT self)

        {

        PS3C4510_CONTEXT        padpt    = &G_AdapterContext;

        TTUSB_PRINTF(((PUDEVICE) self->pDevice,UDMASK_ANY,"+S3C4510_AdapterAttach()\n"));

        TTUSB_DEBUG(UHIL_FlushChar(self->pPlatform);)

        G_AppContext = self; /* for debugging */
```

```
self->pAdptSwitch        = &G_Adapter_switch;
self->pAdptContext       = padpt;
padpt->ReceiveQueue      = NULL;
padpt->pAppContext       = self;
padpt->enabled   = FALSE;
UHIL_fill(&padpt->UsbSynchBlock, sizeof(UHIL_SYNCHRONIZATION_BLOCK), 0);
padpt->ReceiveHalted     = FALSE;
PowerOnReset_LXT971_PHY();       /* init the PHY chip */
LXT971_Enable_External_Int(padpt);
TTUSB_PRINTF(((UDEVICE *)self->pDevice,UDMASK_ANY,"Phy has been reset\n"));
TTUSB_DEBUG(UHIL_FlushChar(self->pPlatform);)
/* Open interrupt connection handles for the rx and tx interrupt */
TTUSB_PRINTF(((PUDEVICE) self->pDevice,UDMASK_ANY,"Open up interrupt connection handles\n"));
G_hTxIntConn = self->pPlatform->upf_pInterruptSystem->pOpenInterruptConnection(
                                self->pPlatform->upf_pInterruptSystem,
                                G_hTxInt
                                );
G_hRxIntConn = self->pPlatform->upf_pInterruptSystem->pOpenInterruptConnection(
                                self->pPlatform->upf_pInterruptSystem,
                                G_hRxInt
                                );
/* setup the DMA interrupts */
G_hBDMARxIntConn = self->pPlatform->upf_pInterruptSystem->pOpenInterruptConnection(
                                self->pPlatform->upf_pInterruptSystem,
                                G_hBDMARxInt
                                );
G_hBDMATxIntConn = self->pPlatform->upf_pInterruptSystem->pOpenInterruptConnection(
                                self->pPlatform->upf_pInterruptSystem,
                                G_hBDMATxInt
                                );
TTUSB_PRINTF(((UDEVICE *)self->pDevice,UDMASK_ANY,"Interrupt connections are open\n"));
TTUSB_DEBUG(UHIL_FlushChar(self->pPlatform);)
/*
|| Now that we have a connection handle, we can set the ISR.  We
|| don't need to check whether the call to pOpenInterruptConnection
|| failed, because pConnectToInterrupt will check for null
|| handles and return FALSE if such are provided.
*/
if (! self->pPlatform->upf_pInterruptSystem->pConnectToInterrupt(
              /* connection handle */  G_hTxIntConn,
              /* ISR function */       MacTxIsr,
              /* the ISR context */    padpt
              ))
              {
              /* connection failed.  Print message */
              TTUSB_PRINTF(((UDEVICE *)
                      self->pDevice,
                      UDMASK_ANY,
                      " ThisRoutine: pConnectToInterrupt failed for MacTxIsr.\n"
                      ));
               UHIL_FlushChar(self->pPlatform);
               /* in this example, we just give up */
               UHIL_swc(UHILERR_INIT_FAIL);
               }
TTUSB_PRINTF(((UDEVICE *)self->pDevice,UDMASK_ANY,"Connected to MacTxIsr int\n"));
TTUSB_DEBUG(UHIL_FlushChar(self->pPlatform);)
if (! self->pPlatform->upf_pInterruptSystem->pConnectToInterrupt(
              /* connection handle */  G_hRxIntConn,
              /* ISR function */       MacRxIsr,
              /* the ISR context */    padpt
              ))
              {
              /* connection failed.  Print message */
              TTUSB_PRINTF((
                      (UDEVICE *)self->pDevice,
                      UDMASK_ANY,
                      " ThisRoutine: pConnectToInterrupt failed for MacRxIsr.\n"
                      ));
               UHIL_FlushChar(self->pPlatform);
               /* in this example, we just give up */
               UHIL_swc(UHILERR_INIT_FAIL);
               }
TTUSB_PRINTF(((UDEVICE *)self->pDevice,UDMASK_ANY,"Connected to MacRxIsr int\n"));
TTUSB_DEBUG(UHIL_FlushChar(self->pPlatform);)
if (! self->pPlatform->upf_pInterruptSystem->pConnectToInterrupt(
              /* connection handle */  G_hBDMATxIntConn,
              /* ISR function */       BDMATxIsr,
              /* the ISR context */    padpt
              ))
              {
              /* connection failed.  Print message */
              TTUSB_PRINTF((
                      (UDEVICE *)self->pDevice,
                      UDMASK_ANY,
                      " ThisRoutine: pConnectToInterrupt failed for BDMATxIsr.\n"
                      ));
               UHIL_FlushChar(self->pPlatform);
               /* in this example, we just give up */
               UHIL_swc(UHILERR_INIT_FAIL);
               }
TTUSB_PRINTF(((UDEVICE *)self->pDevice,UDMASK_ANY,"Connected to BDMATxIsr int\n"));
TTUSB_DEBUG(UHIL_FlushChar(self->pPlatform);)
if (! self->pPlatform->upf_pInterruptSystem->pConnectToInterrupt(
              /* connection handle */  G_hBDMARxIntConn,
              /* ISR function */       BDMARxIsr,
              /* the ISR context */    padpt
              ))
              {
              /* connection failed.  Print message */
              TTUSB_PRINTF((
                      (UDEVICE *)self->pDevice,
                      UDMASK_ANY,
                      " ThisRoutine: pConnectToInterrupt failed for BDMARxIsr.\n"
                      ));
               UHIL_FlushChar(self->pPlatform);

                /* in this example, we just give up */
               UHIL_swc(UHILERR_INIT_FAIL);
               }
TTUSB_PRINTF(((UDEVICE *)self->pDevice,UDMASK_ANY,"All interrupts have been connected to\n"));
TTUSB_DEBUG(UHIL_FlushChar(self->pPlatform);)
/* Init_MAC(self);  */
TTUSB_PRINTF(((PUDEVICE) self->pDevice,UDMASK_ANY,"-S3C4510_AdapterAttach()\n"));
TTUSB_DEBUG(UHIL_FlushChar(self->pPlatform);)
/* Disable MAC and BDMA interrupts. */
self->pPlatform->upf_pInterruptSystem->pInterruptControl(G_hRxIntConn,FALSE);
self->pPlatform->upf_pInterruptSystem->pInterruptControl(G_hTxIntConn,FALSE);
self->pPlatform->upf_pInterruptSystem->pInterruptControl(G_hBDMARxIntConn,FALSE);
self->pPlatform->upf_pInterruptSystem->pInterruptControl(G_hBDMATxIntConn,FALSE);
return TRUE;
}
```

Appendix D – References

Document References

USB Specification Documentation

Documentation Subject	Web Hyperlinks to USB Specifications
Universal Serial Bus Revision 3.0 specification	http://www.usb.org/developers/docs/usb_30_spec_060910.zip
Universal Serial Bus Revision 2.0 specification	http://www.usb.org/developers/docs/usb_20_040908.zip
Common Class Base Specification 1.0	http://www.usb.org/developers/devclass_docs/usbccs10.pdf
Mass Storage Overview 1.1	http://www.usb.org/developers/devclass_docs/usb_msc_overview_1.2.pdf
Mass Storage Bulk Only 1.0	http://www.usb.org/developers/data/devclass/usbmassbulk_10.pdf
Mass Storage Control/Bulk/Interrupt (CBI) Specification 1.0	http://www.usb.org/developers/devclass_docs/usbmassbulk_10.pdf

Acronyms and Abbreviations

Terms	Description
ACM	Abstract Control Model Protocol.
Application	Refers to the code and overall purpose of the capability of the Target Hardware. Applications call protocol software and/or hardware interface drivers to accomplish the main task (application) of the Target Hardware.
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.
cdcether	MCCI implementation of the USB Communication Device Class (CDC) specific for use with networking interfaces.
cross compile	The act of compiling source code on one computer (a Host Computer) that will operate on a different computer (or Target Hardware). This is a required practice for developers developing software/firmware for

	embedded hardware devices.
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.
firmware	A type of software written for an embedded hardware device usually stored in a non-volatile form such as in ROM, EPROM, or flash memory.
HCD	Host Controller Driver. The software component that provides low–level access to the specific Host Controller in use
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 chip.
HNP	Host Negotiation Protocol
Host / Host [Development] Computer	The computer where the DataPump development software and tools are loaded and the application is created. Once the application is created, the executable is moved to the Target or Final Hardware.
loop back [application]	The DataPump software comes with a supplied application/protocol that simply has the Target Hardware echo back whatever is sent to it. Loop back is used to verify correct installation, to instruct on the use of the Data Pump software, and to validate that hardware in operating properly.
MIB	Management Information Base.
MSC	Mass Storage device Class.
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 usually refers to the software and specification of the use of the USB bus for a particular device type. The CDCether Ethernet Control Model (ECM), for Ethernet-like networking, is an example of a protocol across USB.
SIC	Still Image Capture Class – the USB class specification that specifies standard ways of implementing a still-image-capture class device.
SNMP	Simple Network Management Protocol.
Target	Normally refers to the combination of Target Hardware and Target Software/Firmware.
Target CPU	The central processing unit (CPU) that runs the DataPump software once

	loaded onto 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 name referred to in general where the DataPump software will operate. This is different that the Host Development Computer. The Target Hardware becomes the Final Hardware at some point during the development process.
Target Operating System	The operating system that runs the DataPump software once loaded onto the target hardware.
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	Universal Serial Bus 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.

Appendix E – List of Tables & Figures

List of Tables

List of Figures

Appendix F – Revision History

Document Title:	AN_402 MCCI USB DataPump User Guide
Document Reference No.:	BRT_000123
Clearance No.:	BRT#093
Product Page:	http://brtchip.com/product/
Document Feedback:	Send Feedback

Revision	Changes	Date
1.0	Initial release	2017-09-13