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

MCCI USB DataPump Porting Guide

Engineering Report 950000137
Rev. C
Date: 2011/09/24

PROPRIETARY NOTICE AND DISCLAIMER

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

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

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

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

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

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

Document Release History

Rev. A	2003/07/03	Original release
Rev. B	2011/04/04	Changed all references to Moore Computer Consultants, Inc. to MCCI Corporation. Changed document numbers to nine digit versions. DataPump 3.0 Updates
Rev. C	2011/09/24	Added source code disclaimer.

TABLE OF CONTENTS

LIST OF TABLES

1. Introduction

This document provides technical training on the internal details of the architecture and implementation of the MCCI USB DataPump. The suggested audience is engineers who have to understand, update, and debug the DataPump core, and especially for engineers who have to:

- move the DataPump to a new platform
- add support for a new USB device chip to the DataPump.

1.1 Related Documents

In order to understand this document, it's necessary to also study the following documents:

DPREF] *MCCI USB DataPump User's Guide*, MCCI Engineering Report 950000066

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

Also interesting, but not as important, is 950178-3 (MPC 850 Based Cable Modem Porting Guide) – it gives perspective on how the DataPump is initialized for a given application.

Code that should be reviewed:

- usbkern/app/rndisbrg – this is a full application, which shows how the V2.01 initialization tables are used. This is especially interesting because it shows how two USB device interfaces are handled (the applications is a USB-to-USB bridge).

- usbkern/app/vspdemo – another full application, that uses the "virtual serial port" in loop-back mode.

- usbkern/arch/arm7/port/wombat/common/upfwombat.c – the initialization code for Wombat – this is the only code for Wombat that "knows" that the wombat has a USS820. If you're modifying the wombat to have a different chip, you'll need to modify upfwombat.c.

2. Notation

2.1 DataPump Porting Terminology

Each port is a combination of a specific:

1. Chip

2. CPU Architecture

3. Target compiler
4. Target board/system
5. Target operating system
6. Development environment
7. Make Utility – now, exclusively BSDmake, but you will see historical evidence of support for other make utilities
8. Target application (what the device does)

In a sense the target "project" defines all these things. We use the notation "port" to indicate a specific combination of the above.

Normally, we try to separate the "application" from the "DataPump" itself. In theory the application should be portable, but normally there are a number of additional platform interfaces.

For example: the USS820 has the following ports already in our tree:

Table 1. Some Supported Chips in the MCCI Tree

Chip	Target Board	CPU	Compiler	Debug	Target OS	Development Environment	Make
USS820	Piglet	68302	GCC	GDB	os/none	UNIX or Windows	BSDmake
USS820	Piglet	68302	SDS 7.1	SDS	os/none	Windows	BSDmake
USS820	Crossfire	68302	SDS 7.1	SDS	"Telenet works"	Windows	PolyMake4
USS820	ISA Eval (obsolete)	80x86 real mode	MS C1.5 (16 bit)	SoftIce (16-bit)	DOS (essentially none)	Windows	NMAKE
USS820	Catena 1610	80x86 protect mode	MS Visual C V6	SoftIce (32 bit)	Windows	Windows	BSDmake
BCM3310	"Empress"	R3000	Diab	printf	pSOS 2.5	Windows	BSDmake (libraries); MKSmake (final app)
MPC850	goldeneye	PPC	GCC	GDB	IOS	UNIX or Windows	BSDmake
SA1110	Ibex	SA1110	Greenhill	Greenhill	ThreadX	Windows or Unix (test), Windows (for customer)	BSDmake

Chip	Target Board	CPU	Compiler	Debug	Target OS	Development Environment	Make
D12	none (ISA support is out of date)	68K	GCC	GDB	none	Windows or Unix	BSDmake
DBX	IPC	MCORE	GCC	GDB	TrueTask	Windows	BSDmake
CL2162	Costalis	Sparc	GCC	GDB	Nucleus	Windows	GNUmake
ISP1362	Catena 1620	80x86 protect mode	MS Visual C V6	MSVS	Windows	Windows	BSDmake

3. USB Fundamentals

This document assumes that the audience has a good fundamental understanding of USB 1.1 technology, as provided by Paul Berg's class, or equivalent.

4. DataPump Overview

For an architectural overview, please refer to the DataPump architectural overview in DataPump User's Guide, [DPREF]. The rest of this section presents additional details about the implementation.

4.1 Replaceable Routines

The DataPump is delivered as a family of libraries. The porting engineer is responsible for supplying a number of low-level routines (outlined in section 5). In most cases, a default version of this routine is supplied as part of the library. In a few cases, these routines cannot be provided in a generic form, and thus the porting engineer must provide these routines.

Starting with V1.61, the philosophy of replaceable routines has been changed. We no longer have replaceable (called-by-name) routines in the libraries. Instead, the canonical-name routines use a method in the (expanded) UPLATFORM table to do their work; and a default implementation is provided in the library (with a name of form UHIL _xxx_Default).

4.2 Function Naming Conventions

The DataPump is lamentably inconsistent about naming conventions. Generally speaking, though, the following rules are followed:

- Usb*Name*: the routine is part of the core DataPump (the portable layer).

- UHIL_xxx: the routine is a replaceable routine that is intended to provide platform-specific abstract functions for the DataPump core.

- UPF_xxx: the routine is a platform-specific routine, which might not be present on all platforms (or might not be used by all chips, or might not be used at all, depending on implementation). *Note, though, that UPF_GetEventContext()* is really a *UHIL-like* routine by this definition, as it is needed if you use the standard UHIL routines from the library.

- *Chipprefix_xxx*: the routine is an internal routine that is specific to a given chip.

5. Hardware Interface Layer (HIL)

5.1 Structures defined by the HIL

5.1.1 The UPLATFORM structure

The UPLATFORM structure is used for collecting information that is specific to the operating system and system hardware in use. One UPLATFORM can be shared by many UDEVICEs (although there is no requirement that there is one and only one UPLATFORM structure).

The UPLATFORM structure contains all the method routines that connect us to the local operating system with the following notable exceptions:

1. UHIL_setpsw
2. UHIL_di

These are left as call-by-name functions for efficiency's sake.

5.2 Services needed by Chip Drivers

5.2.1 Hardware I/O macros

Hardware I/O macros are used for accessing hardware registers. We define basic I/O macros for each of the basic data types:

UHIL_<op>b	for UINT8 access
UHIL_<op>w	for UINT16 access
UHIL_<op>d	for UINT32 access

The <op> represent read or write operation; "in" means read, and "out" means write operation. The names other than "b" follow the Intel convention that w == word == 16 bits and d == dword == 32 bits.

To allow a calling package to selectively override, we check for each definition whether the macro UHIL_<op><width>_defined is defined; if so, we assume that the corresponding symbol UHIL_<op><width> is (or will be) defined elsewhere. By using a separate symbol, we provide for situations in which UHIL_<op><width> must be a function rather than a macro.

5.2.2 Interrupt Service Routine (ISR) declaration and dispatching

The interrupt service routine is called in response to an interrupt on hardware IRQ. It is passed a context pointer, specified when the routine was registered. In order to support shared interrupt schemes (like PCI), the ISR is required to return TRUE if it thinks that its device was interrupting, and FALSE otherwise. The platform is, of course, at liberty to ignore this result.

The interrupt service routine is defined as:

```
typedef BOOL (UHIL_INTERRUPT_SERVICE_ROUTINE_FN)(VOID *);
```

5.2.3 DMA Support Abstractions (Optional)

5.3 Services needed by the DataPump Core

The implementation of the following services is normally specific to a particular combination of: CPU, compiler, and target OS. However, it normally is not affected by the choice of chip.

5.3.1 Memory allocation

In the version 1.61 or higher DataPump, any memory that is dynamically allocated must be allocated during initialization.

General purpose memory is allocated using a primitive provided in the UPLATORM structure; it's freed using another primitive from the UPLATFORM structure.

Device-specific memory is allocated using the single primitive:

```
VOID *UsbAllocateDeviceBuffer(
            UDEVICE *pDevice, /* the device itself */
            BYTES Size        /* the size of the buffer */
            );
```

Device-specific memory is freed using the primitive:

```
VOID UDEVFREE(
            UDEVICE *pDevice, /* the device itself */
            VOID *pBuffer,    /* allocated buffer */
            BYTES Size        /* the size of the buffer */
            );
```

Note that UDEVFREE() is a macro that used the device switch directly.

These routines are virtual – it invokes a method in the UDEVICESWITCH. A subroutine is provided in the common library (common/devbuf.c) which can be used as the default if no better implementation is available.

The abstract semantics of this routine are:

`UsbAllocateDeviceBuffer()` allocates a buffer that is *associated* with the specified device. These buffers are simply memory, allocated as it were by `malloc()`, but they have the additional special property: they can be used as buffers that are passed as part of UBUFQEs.

Some USB device hardware uses DMA, but the DMA hardware has certain restrictions about how data must be passed to that hardware. For example, the buffers might have to reside within a single 64K block of memory; or the buffers might have to satisfy certain alignment constraints. `UsbAllocateDeviceBuffer()` must allocate buffers that satisfy the device's restrictions whatever they are.

5.3.2 Interrupt System Control

5.3.2.1 UHIL_di

```
CPUFLAGS UHIL_di(VOID);
```

The CPU status word is modified so that interrupts are disabled. A platform-specific representation of the CPU status word before it was modified. It is assumed that the caller will save this value and use it as the argument to UHIL_setpsw() to restore the previous value. Portable code must treat this as opaque data. Note that MCCI's code disables interrupts sparingly, and does not attempt to take advantage of the multiple priority levels available on some CPUs.

5.3.2.2 UHIL_setpsw

```
VOID UHIL_setpsw(
      ARG_CPUFLAGS psw
      );
```

The argument `psw` is written into the CPU status word to restore interrupts to their previous value. Users of this function should never make any assumptions about the meanings of various bits in CPUFLAGS or ARG_CPUFLAGS.

5.3.3 Synchronization / Event queue processing

5.3.3.1 UHIL_SynchronizeToDataPumpDevice

```
VOID UHIL_SynchronizeToDataPumpDevice(
      UDEVICE *pDevice,
      UHIL_SYNCHRONIZATION_BLOCK *pSynchBlock,
      UHIL_SYNCHRONIZED_FN *pSynchFn,
      VOID *Context1,
      VOID *Context2
      );
```

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

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

5.3.3.2 UHIL_PostEvent

```
VOID UHIL_PostEvent(
      PUEVENTCONTEXT pevq,          /* event queue */
      CALLBACKCOMPLETION *ctx       /* callback context */
      );
```

We start by making sure that interrupts are disabled so as to prevent reentrancy. We then save the event information in the event queue at the slot pointed to by the queue tail pointer.

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

If advancing tail will cause tail to point to the same cell as head, we do not advance tail and instead increment an overrun counter.

5.3.4 Miscellaneous library routines

5.3.4.1 UHIL_cpybuf

```
BYTES UHIL_cpybuf(
      VOID *pDest,            /* the desination buffer */
      CONST VOID *pSrc,       /* the source buffer */
      BYTES Size              /* the size of the buffer */
      );
```

This is a `memcpy()`-like function. The DataPump uses this instead of `memcpy()` in order to avoid headaches on systems that are "almost ANSI" – a surprising number of compiler libraries for embedded work are not quite ANSI, even though the compiler itself is.

Note that `UHIL_cpybuf()` returns `Size`. Also, `UHIL_cpybuf()` checks that source and destination pointers are both not `NULL`.

On some ports, this function is hand-optimized to copy multiples of 64 bytes efficiently.

Finally, `UHIL_cpybuf()` must work correctly even if the buffers overlap.

5.3.4.2 UHIL_fill

```
BYTES UHIL_fill(
      VOID *pDest,            /* the desination buffer */
      ARG_BYTES Size          /* the size of the buffer */
      ARG_UCHAR FillValue     /* the value to write */
      );
```

This is a `memset()`-like function. The DataPump uses this instead of `memset()` in order to avoid headaches on systems that are "almost ANSI" – a surprising number of compiler libraries for embedded work are not quite ANSI, even though the compiler itself is.

Note that `UHIL_fill()` has a different argument order than `memset()`. It returns `Size`. Also, `UHIL_fill()` checks that the destination pointer is not `NULL`.

6. How Chips are Connected Up

6.1 The Upper Edge

6.1.1 Entry points: the UDEVICESWITCH

All calls from the DataPump core to chip drivers are made via the UDEVICESWITCH, a data structure which contains pointers to functions. The Hardware Interface Layer (or miniport driver for the USB peripheral interface silicon) exports nine functions to the USB DataPump.

One instance of this switch is exported for each kind of USB chip.

The UDEVICESWITCH is defined as:

```
typedef USTAT (UDEVINITIALIZEFN)(
            CONST UDEVICESWITCH *   pSwitch,
            UDEVICE *               pSelfPortable,
            BYTES                   nSelfPortable,
            UPLATFORM *             pPlatform,
            UHIL_BUSHANDLE          hBusHandle,
            IOPORT                  IoPort,
            UDEVICE_INITFN          pInitFn,
            CONST USBRC_ROOTTABLE * pRoot,
            UINT32                  DebugFlags,
            CONST VOID *            pConfigData,
            BYTES                   nConfigData
            );
typedef VOID (UDEVSTARTFN)(UDEVICE *);
typedef VOID (UDEVSTOPFN)(UDEVICE *);
typedef VOID (UDEVREPORTEVENTFN)(
            UDEVICE *         pSelfAbstract,
            ARG_UEVENT        Why,
            VOID *            pEvInfoAbstract
            );
typedef BYTES (UDEVREPLYFN)(
            UDEVICE *         pSelfAbstract,
            CONST VOID *      pBuf,
            BYTES             LenRequested,
            BYTES             LenActual,
            UBUFIODONEFN *    pDoneFn,
            VOID *            pDoneCtx,
            UBUFQE_FLAGS      Flags
            );
typedef VOID (UDEVREMOTEWAKEUPFN)(UDEVICE *pSelfAbstract);
typedef VOID *(UDEVALLOCATEFN)(
            UDEVICE *         pSelfAbstract,
            BYTES             nAllocate
            );
typedef VOID (UDEVFREEFN)(
            UDEVICE *         pSelfAbstract,
            VOID *            pBuf,
            BYTES             nBuf
            );
typedef USBPUMP_IOCTL_RESULT (UDEVIOCTLFN)(
```

```
                UDEVICE *         pSelfAbstract,
                USBPUMP_IOCTL_CODE IoctlCode,
                CONST VOID *      pInParam,
                VOID *            pOutParam
                );

        struct TTUSB_UDEVICESWITCH
              {
              UDEVINITIALIZEFN        *udevsw_Initialize;
              UDEVSTARTFN             *udevsw_Start;
              UDEVSTOPFN              *udevsw_Stop;
              UDEVREPORTEVENTFN       *udevsw_ReportEvent;
              UDEVREPLYFN             *udevsw_Reply;
              UDEVREMOTEWAKEUPFN      *udevsw_RemoteWakeup;
              UDEVALLOCATEFN          *udevsw_AllocateDeviceBuffer;
              UDEVFREEFN              *udevsw_FreeDeviceBuffer;
              UDEVIOCTLFN             *udevsw_Ioctl;
              };
        typedef struct TTUSB_DEVICESWITCH
              UDEVICESWITCH, *PUDEVICESWITCH;
```

We defined some macro to call these functions. Rather than calling functions directly.

To initialize device:

```
        UDEVINITIALIZE(
              /* CONST UDEVICESWITCH *      */    pSwitch,
              /* UDEVICE *                  */    pSelfPortable,
              /* BYTES                      */    nSelfPortable,
              /* UPLATFORM *                */    pPlatform,
              /* UHIL_BUSHANDLE             */    hBusHandle,
              /* IOPORT                     */    IoPort,
              /* UDEVICE_INITFN             */    pInitFn,
              /* CONST USBRC_ROOTTABLE *    */    pRoot,
              /* UINT32                     */    DebugFlags,
              /* CONST VOID *               */    pConfigData,
              /* BYTES                      */    nConfigData
              );
```

To start the device up:

```
        UDEVSTART(
              /* UDEVICE *           */    pSelf
              );
```

To request that the device stop:

```
        UDEVSTOP(
              /* UDEVICE *           */    pSelf
              );
```

To report a device event:

```
UDEVREPORTEVENT(
      /* UDEVICE *            */    pSelf,
      /* ARG_UEVENT           */    Why,
      /* VOID *               */    pEvInfoAbstract
      );
```

To send a reply to the current control transaction:

```
UDEVREPLY(
      /* UDEVICE *            */    pSelf,
      /* CONST VOID *         */    pBuf,
      /* BYTES                */    LenRequested,
      /* BYTES                */    LenActual,
      /* UBUFIODONEFN *       */    pDoneFn,
      /* VOID *               */    pDoneCtx,
      /* UBUFQE_FLAGS         */    Flags
      );
```

To request a remote wakeup:

```
UDEVREMOTEWAKEUP(
      /* UDEVICE *            */    pSelf
      );
```

To allocate a packet that is guaranteed OK for data transfer:

```
UDEVALLOCATE(
      /* UDEVICE *            */    pSelf,
      /* BYTES                */    nAllocate
      );
```

To release a packet allocated by UDEVALLOCATE:

```
UDEVFREE(
      /* UDEVICE *            */    pSelf,
      /* VOID *               */    pBuf,
      /* BYTES                */    nBuf
      );
```

To control device:

```
UDEVIOCTL(
      /* UDEVICE *            */    pSelf,
      /* USBPUMP_IOCTL_CODE   */    IoctlCode,
      /* CONST VOID *         */    pInParam,
      /* VOID *               */    pOutParam
      );
```

6.1.2 Shared Extensible Data Structures

Most MCCI USB data types are constructed in a manner that allows other types to be derived from them. This allows the types to be extended as needed. Such extensions are typically needed to allow a hardware-specific driver to attach hardware-specific state information to a particular data instance. MCCI prefers this approach to the "device extension" approach because all the data items are contiguous, allowing for simplified MMU support, and allowing various levels of the system to communicate using the same data object. The Derived Data types

also allow for stronger type checking in the driver sources because ambiguous pointer types are not used.

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

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

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

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

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

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

struct MBASETYPE
      {
      MBASETYPE_HDR;
      };

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

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

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

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

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

6.1.2.1 UDEVICE

This structure represents a single USB device to the USB DataPump.

Typedef: UDEVICE, *PUDEVICE

Embedding Macro: UDEVICE_HDR

```
/**** the contents of the embedded header ****/
struct TTUSB_UDEVICE_HDR
      {
      USBPUMP_OBJECT_HEADER   udevhh_Header;
      CONST USBRC_ROOTTABLE * udevhh_pDescriptorRoot;
      CONST USBIF_DEVDESC_WIRE *udevhh_pDevDesc;
      UINT16                  udevhh_usbPortIndex;
      UINT8                   udevhh_usbDeviceStatus;
      UINT8                   udevhh_usbDeviceAddress;
      UINT32                  udevhh_usbDeviceEnumCounter;
      UCONFIG     *           udevhh_pAllConfigs;
      UCONFIG     *           udevhh_pConfigs;
      UCONFIG     *           udevhh_pCurrent;
      VOID        *           udevhh_pExtension;
      UPLATFORM   *           udevhh_pPlatform;
      CONST UDEVICESWITCH *   udevhh_pSwitch;
      UEVENTNODE  *           udevhh_noteq;
      UINTERFACESET *         udevhh_pCtrlIfcset;
      UENDPOINT   *           udevhh_pEndpoints;
      UINT8       *           udevhh_pReplyBuf;
      USBPUMP_ABSTRACT_POOL   *           udevhh_pPoolHead;
      UINTERFACESET *         udevhh_pvAllInterfaceSets;
      UINTERFACE  *           udevhh_pvAllInterfaces;
      UPIPE       *           udevhh_pvAllPipes;
      SIZE_T                  udevhh_sizeDevPool;
      ADDRBITS                udevhh_alignmentDevPool;
      BYTES                   udevhh_sizeReplyBuf;
      UINT16                  udevhh_wNumAllPipes;
      UINT8                   udevhh_bCurrentSpeed;
      UINT8                   udevhh_bSupportedSpeeds;
      UINT8                   udevhh_RemoteWakeupEnable;
      UINT8                   udevhh_L1RemoteWakeupEnable;
      UINT8                   udevhh_LinkState;
      UINT8                   udevhh_fSuspendState;
      UINT16                  udevhh_nAutoRemoteWakeup;
      UINT8                   udevhh_bTestMode;
      UINT8                   udevhh_fLpmEnable;
      UINT8                   udevhh_HnpEnable;
      UINT8                   udevhh_HnpSupport;
      UINT8                   udevhh_AltHnpSupport;
      UINT16                  udevhh_wNumAllConfigs;
      UINT8                   udevhh_bNumConfigs;
      UINT8                   udevhh_bNumHSConfigs;
      UINT8                   udevhh_bNumEndpoints;
      UINT8                   udevhh_bNumAllInterfaceSets;
      UINT8                   udevhh_bNumAllInterfaces;
      UINT8             udevhh_bActiveConfigurationValue;
      UINTERFACESET           udevhh_ctlifcset;
      UINTERFACE              udevhh_ctlifc;
      UBUFQE                  udevhh_ctlsetupbq;
      UBUFQE                  udevhh_ctlinbq;
      UBUFQE                  udevhh_ctloutbq;
      USETUP_HANDLE           udevhh_hSetup;
      };
```

udevhh_Header	the header of device.
udevhh_pDescriptorRoot	the root table pointer.
udevhh_pDevDesc	the pointer to our device descriptor.
udevhh_usbPortIndex	the USB mib port index, but zero-origin; fields were added in V1.60n, but support is incomplete.
udevhh_usbDeviceStatus	the current device status.
udevhh_usbDeviceAddress	the current device address.
udevhh_usbDeviceEnumCounter	count of 'enumerations'
udevhh_pAllConfigs	the vector of known configurations for all speed.
udevhh_pConfigs	the vector of known configurations for this device.
udevhh_pCurrent	the current configuration or is set to NULL to indicate that the device is not currently configured.
udevhh_pExtension	is an extension pointer to application-specific data.
udevhh_pPlatform	the platform for this device.
udevhh_pSwitch	is the switch structure that provides the functional interface for the DataPump to communicate with the HIL for device level interactions.
udevhh_noteq	is the event notification queue for events attached at the device level.
udevhh_pCtrlIfcset	the default ifcset.
udevhh_pEndpoints	an array of UENDPOINT structures.
udevhh_pReplyBuf	the reply buffer.
udevhh_pPoolHead	the current head of the device pool.
udevhh_pvAllInterfaceSets	the vector of all interface sets for the device.
udevhh_pvAllInterfaces	the vector of all interfaces.
udevhh_pvAllPipes	the vector of all pipes.
udevhh_sizeDevPool	the total size of the device pool.
udevhh_alignmentDevPool	the alignment of the device pool.

udevhh_sizeReplyBuf	the size (capacity) of the reply buffer.
udevhh_wNumAllPipes	the number of pipes, total.
udevhh_bCurrentSpeed	the device speed.
udevhh_bSupportedSpeeds	the device speed set.
udevhh_RemoteWakeupEnable	is Boolean. Setting it to TRUE will enable remote wakeup.
udevhh_L1RemoteWakeupEnable	is Boolean. Setting it to TRUE will enable L1 remote wakeup.
udevhh_LinkState	the link state.
udevhh_fSuspendState	is Boolean. Setting it to TRUE means suspend state.
udevhh_nAutoRemoteWakeup	counter for automatic remote wakup feature.
udevhh_bTestMode	if non-zero, the most recently-selected test mode.
udevhh_fLpmEnable	Setting it to TRUE will enable LPM.
udevhh_HnpEnable	Setting it to TRUE will enable HNP.
udevhh_HnpSupport	Setting it to TRUE means HNP is supported on this port.
udevhh_AltHnpSupport	Setting it to TRUE means HNP is supported somewhere on this host.
udevhh_wNumAllConfigs	the number of all possible configurations.
udevhh_bNumConfigs	the number of possible configurations for this device.
udevhh_bNumHSConfigs	the number of HS configurations.
udevhh_bNumEndpoints	the number of endpoints for this device.
udevhh_bNumAllInterfaceSets	the length of vAllInterfaceSets.
udevhh_bNumAllInterfaces	the length of vAllInterfaces.
udevhh_bActiveConfigurationValue	the active configuration value.
udevhh_ctlifcset	the interface set for the control interface.
udevhh_ctlifc	the control interface.
udevhh_ctrlsetupbq	the setup buffer queue for the control interface.

udevhh_ctlinbq the in buffer queue for the control interface.

udevhh_ctloutbq the out buffer queue for the control interface.

udevhh_hSetup the setup handle.

6.1.2.2 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

```
/**** the contents of the embedded header ****/
struct TTUSB_UENDPOINT_HDR
      {
      UBUFQE *    uephh_pending;
      UPIPE *     uephh_pPipe;
      VOID *      uephh_pExtension;
      CONST UENDPOINTSWITCH *uephh_pSwitch;
      UINT        uephh_stall: 1,
                  uephh_fChanged: 1;
      UCHAR       uephh_Size;
      UCHAR       uephh_siolock;
      UCHAR       uephh_ucTimeoutFrames;
      };
```

uephh_pending the queue of UBUFQE structures being filled.

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

uephh_pExtension an application-specific extension data area if needed.

uephh_pSwitch the switch structure that provides the functional interface for the DataPump to communicate with the HIL for endpoint level interactions.

uephh_stall set to TRUE whenever the endpoint stalls.

uephh_fChanged configuration of endpoint changed.

uephh_Size the size of this structure.

uephh_siolock the start-I/O lock-out count.

uephh_ucTimeoutFrames the timeout down-counter.

6.2 The Lower Edges

6.2.1 Talking to Chip Hardware

Normally, a chip will have a *base address* in some *address space*. Sometimes the address space is system memory address space; this means that you can use pointer operations to talk to the chip. Sometimes the address space is some system I/O address space; this means that you must use special (compiler- and possibly CPU-specific operations) to talk to the chip. Unfortunately, there normally is no way to predict whether a given chip will be "memory mapped" or "I/O mapped"; the system designer decides.

So, we have a means of hiding this detail from the person who codes the chip-specific code. All our HILs have implementations of the following macros:

- UHIL_inb(), UHIL_inw(), UHIL_ind() – read a byte, word, or dword (32 bits) from a device
- UHIL_outb(), UHIL_outw(), UHIL_outd() – write a byte, word or dword
- UHIL_insb(), UHIL_insw(), UHIL_insw() – read a string of bytes, words or dwords
- UHIL_outsb(), UHIL_outsw(), UHIL_outsd() – write a string of bytes words, or dwords..

These macros are *abstract* APIs that take the following parameters:

- A *bus handle* – this is an opaque type that represents (to the HIL) what bus the device is on. In many HIL *implementations*, this bus handle is ignored, for example if we know in advance that the all devices are memory mapped.
- A *port number* – this is an unsigned integral type that represents a location on that bus.

These (bus handle, port number) pairs are primitive – and somewhat hard to use. Chips are normally accessed via a set of port numbers that are related to a base address. Unfortunately, the exact relationship tends to vary from platform to platform. When an 8-bit device is used on an 8-bit platform, the registers may be separated from each other by increments of 1. On a 32-bit platform, the registers might be separated from each other by increments of 4. In addition, we normally want to write our chip code so we can support multiple instances of a given chip. And we want to support plug-and-play environments in which the base port address is not known until run-time.

On the other hand, in most embedded environments, the chip base address and access methods are fixed for all time when the board is designed. Too much generality can be inefficient and slow.

Therefore, the chip "include" files normally impose an additional layer of abstraction. We define chip-specific macros that are implemented in general ways in terms of the UHIL_xxx primitives, but which can further be overridden by port-specific code, if needed. Because these macros are normally defined in the chip-specific data structure files, we normally define these to a pointer to the chip instance structure as their first argument, and a register name as the second argument.

In addition, these chip-specific macros often take care of unpleasant or inconvenient aspects of the low-level API to the chip. For example, for heavily multiplexed chips, we try to hide the multiplexing in the macros.

On the USS820, we have:

- USS820GET() and PUT(), which move single bytes to registers
- USS820GETSB() and PUTSB(), which move multiple bytes to a given register. These are in fact where all the time is spent when moving data, so these are critical to the implementation.
- USS820PUT_ANDOR(), which does a read/modify/write cycle
- USS820PUT_PEND_ANDOR(), which uses the USS820 PEND register to do an interlocked read/modify/write.

One thing to note is that all of these operations are NOT primitive, and are NOT expected to apply to any other chip – they were defined solely with the convenience of the chip-specific code in mind, and also to allow special applications of the USS820 to use the same source code as the normal applications.

On the D12, we need to worry about different details.

1. Instead of registers, the D12 has commands and responses. Data is accessed by writing a command select code to the address port, and then by reading or writing the data port.
2. The address-port is write-only, and has to be modified by the ISR, in order to clear the interrupt pin (worst case – depending on the platform, this might be avoidable). This means that we must have some way of protecting the background code (also using the address port) from being clobbered if an interrupt occurs.
3. Most registers are read-only or write-only. This means that we don't have to worry about read-modify-write operations to the hardware. The port logic will be responsible for keeping shadow values of key registers in RAM, as needed. (This data will be stored in the per-chip instance-data structure.)

6.2.2 Interrupts

The HIL exports two key functions.

CPUFLAGS UHIL_di(VOID);

VOID UHIL_setpsw(CPUFLAGS SavedFlags);

UHIL_di() is used to disable all interrupts on the system. It returns an opaque value, which must be saved by the caller. Later, when the caller wants to re-enable interrupts, the caller should pass that value to UHIL_setpsw().

This definition allows UHIL_di/UHIL_setpsw() to be used in pairs. We assume that these pairs can be nested – i.e., x = UHIL_di(); … y = UHIL_di(); UHIL_setpsw(y); … UHIL_setpsw(x); will function in the expected way, disabling interrupts at the first call to UHIL_di, and re-enabling at the last call.

These primitives must be used sparingly. It is OK to use these to interlock accesses to the chip hardware, for chips like the D12 which are command based, and which may need to talk to the chip in the primary ISR. It is not OK to use these to disable interrupts for longer than a few microseconds.

In many implementations, these are implemented as subroutine calls. This means that these are relatively expensive primitives. If your chip needs fast versions, you should consider modifying the HIL for that platform to export these as inline-assembly macros.

6.3 Chip Instance Data

The design of the DataPump is based on a per-USB-chip data structure. This data structure is divided into three sections:

1. The portable part contains data that is neither specific to the chip nor to the application. This part is called a UDEVICE, and is always at the beginning of the actual data structure. There never, in fact, is a bare UDEVICE; UDEVICEs are always at the beginning of a more complete, concrete chip- and application-specific data structure.

2. The chip-specific part starts with a UDEVICE (so it's legal to convert a chip-specific pointer to a UDEVICE with a simple cast). It then continues with data that's specific to that chip. For example, a UDEV820 is the instance data for any USS820; a UDEV3310 is the instance data for a Broadcom 3310. Note well that we use this abstraction even on ports (such as the 3310) in which only one instance is possible. Again, UDEV*xxx* structures are always abstract – they are always part of the application-specific data structure.

3. The application-specific instance data structure is fully concrete. The definition is generated by USBRC from the descriptors for the device and from chip-specific information that is input to USBRC. The type name is chosen by the person who writes the application code and the *.urc* file.

Since all chips are assumed to have bus handles and port offsets, the UDEVICE contains a bus handle slot and a base port slot. Chips that need multiple bus handles and port offsets are free to define additional cells, but these should be put in the chip-specific data structure.

The UDEVICE contains a pointer to a special structure, the "UDEVICE switch". This is the function table that connects portable code to chip specific code. The UDEVICE switch contains pointers to various top-level entry points.

6.3.1 The UENDPOINT structure

The UDEVICE is also the root of the device tree, which models the device. At the bottom of the tree are several UENDPOINT structures, which model the physical endpoints of the device. Just as the UDEVICE has a corresponding UDEV*xxx* structure, the UENDPOINTs are part of chip-specific endpoint-structures. On the USS820, these are called EPIO820's; on the Broadcom 3310, these are called EPIO3310. For future chips, we have adopted a new convention of calling these "UEP*xxx*", for consistency with our other names.

The UENDPOINT contains a pointer to a function table that connects the endpoint to the portable code; this table is called the "Endpoint Switch". Unlike the device switch, this pointer will change at run time, depending on how the endpoint is currently configured. Most important, the switch contains function pointers for queuing UBUFQEs to an endpoint, and for processing "events" that have been received through the chip level code.

The UENDPOINTSWITCH contains, in fact, two kinds of functions. Some functions are called from the upper layers (uepsw_StartIo, uepsw_PrepIo, and uepsw_CancelIo) to perform abstract operations. Others (uepsw_Timeout and uepsw_EpEvent) are available to be called internally (from within the chip-specific code) to perform abstract operations that may need to be handled differently based on how the endpoint is currently configured. However, since these are internal APIs, the chip-specific code is free to take advantage of this facility or not. (In some ways, the UENDPOINTSWITCH should probably been defined as a two-part data structure, so that the chip-level code could do whatever made sense. In particular, the current approach needlessly forces extra casts to be applied, because the portable APIs must be defined in terms of the abstract UDEVICE, but both the endpoint code and the device code for a given chip always operate using a more specific UDEV*xxx*.)

7. D12 Specifics

On the D12 port, we'll call the instance data block a "UDEVD12".

We'll call the endpoint-structure a "UEPD12".

The UDEVD12 needs to contain the following info:

- Most recent address port value
- Saved copies of the internal state
- Additional sequencing information as needed for handling setup packets, etc

The UEPD12 needs to contain:

- the stall-processing buffer
- the index that selects the endpoint (0..5)
- the interrupt mask value (for the DMA-capable endpoints)

It's possible that we'll need additional info, because the DMA controller is external to the D12 – but this has to be tuned on a port-by-port basis. The initial version doesn't support DMA.

7.1 D12 architecture

We'll need:

- A UDEVICESWITCH that dispatches the chip-level operations from the abstract level. The functions are: (Initialize, ReportEvent, Reply, Start, Stop, and RemoteWakeup).

- UENDPOINTSWITCHes to handle control OUT, control IN, bulk OUT, bulk IN, interrupt IN. ISOCH I/O can be ignored for the moment, because our first customer doesn't need it.

- A primary ISR to take device interrupts and dispatch to a secondary ISR via the event mechanism.

- A secondary ISR (scheduled via the event mechanism) that decodes the device interrupts and device state. Device state changes are normally handled directly by the secondary ISR; endpoint state changes are normally dispatched (indirectly) via the UENDPOINTSWITCH specific to that endpoint.

- An initialization function.

7.2 Handling STALLS on the D12

In order to support guaranteed recovery from a write timeout on printer-class devices, the D12 port needs to do some special handling when a SET FEATURE (Endpoint Stall) is received over the control endpoint, targeting a BULK ENDPOINT. The proposed sequence of operations is:

1. Set the endpoint stall flag for the target endpoint. (This should make sure that no more data is delivered into the hardware FIFO.)

2. Drain the FIFOs into a special, additional buffer that's dedicated for this purpose.

3. Complete the "SET FEATURE (Endpoint Stall)" operation (this makes sure the host can't try to do a CLEAR FEATURE until we've drained the buffer.

Then the BULK-OUT code will need to look aside into the FIFO buffer to make sure that the data is transported to the application.

8. How Operating Environments are Supported

A "port" is a given specific target -- it defines the binding between OS, USB chip, CPU (and possibly compiler) for a given build environment.

All ports live under a specific CPU environment, in the arch/{tag}/port/ directory. The arch/arm7/port/ directory contain ARM7 CPU environments.

Each port has its own directory. For each port, a library is created consisting of the various files needed to support the platform; the files are actually linked in based on the underlying OS's startup requirements and code. But since the OS for a given port is somewhat predetermined, no guarantee is currently made for OS-independence of this code.

This section explains how to create a new port for a given CPU architecture of the DataPump.

8.1 Select a name for the port

By convention, this port name is up to 8 characters long, and is chosen from the set of valid DOS file names, and all letters are lower case (wombat, catena, etc.). The lower-case part is not purely convention -- there are issues for cross-platform portability.

8.2 Create the port directory

The port directory is created like arch/{arch}/port/{port} and add it to the CVS tree. The {arch} represents CPU architecture (arm7, i386, real68k, mcore, etc.), and the {port} represents your selected port name. For example, the Wombat's port directory is arch/arm7/port/wombat.

In the {port} directory, create at least the following directories:

common/	the platform initialization codes
i/	the platform header files
mk/	the build environments
packlist/	the release packing list

8.3 Choose your target OS

The DataPump Firmware Development Kits directly supports a variety of firmware based operating systems (including pSOS, ThreadX, Nucleus, PowerTV, OSE, Windows or MCCI's proprietary operating system - TrueTask), and supports also no target operation system.

For OS support code for the platform, create a {port}/os/{osname} directory, and put a UsbMakefile.inc there. The OS library build will find it and include it in the datapump library.

8.4 Create {port}/common

In the {port}/common directory, you create the platform initialization code. If you're using os/none, then this must contain a UPF_main() function and any low-level hardware initialization code. For other operating systems, you must refer to the operating-specific information.

8.5 Create a UsbMakefile.inc

In the {port}/ directory, create an UsbMakefile.inc file. It must point to the files in the {port}/common directory. This is how you arrange for these files to get compiled. Look at the arch/arm7/port/wombat/UsbMakefile.inc for a good example.

8.6 Create {port}/mk

In the {port}/mk directory, create a buildset.var file; you'll need to refer to the arch/{arch}/mk/buildset.var file for exact information on how to do this.

Among the things you must set are the IFC variable, which must select a valid USB DataPump device (uss820, d12, etc.); the OS variable, which selects the operating system; the LIBPORT_TOOLSET setting, which selects the compiler, and possibly other CPU-specific selections.

For example, on Arm-based ports, you should set CPU_MODEL and CPU_ENDIAN to match the setup of the target.

More often than we'd like, you'll also have to add a CFLAGS_PORT in the [makefile] section; this setting should be

```
CFLAGS_PORT += -D__TMS_USE_STDDEF_FOR_SIZE_T=1
```

which tells libport how to handle size_t (otherwise it guesses wrong).

If you want to build the applications under control of bsdmake, you should append values to APPLIST in the [makefile] section. These values are pointers to the top level of the application (actually, to the directory that holds the app's UsbMakefile.inc).

8.7 Create {port}/i

In the {port}/i directory, you'll definitely need a hilbusio.h -- this specifies how the rest of the code will talk to devices. Frequently this points to a common file in a higher-level directory. Sometimes this specifies accelerated I/O methods. Look at arch/real68k/port/piglet/i/hilbusio.h for an example of how to use DMA to load/unload the FIFOs of a USS820.

8.8 Create {port}/packlist

You make a {port}/packlist/source-release.pkl file that specifies how this portion of the tree is to be exported.

8.9 Create README

Please put a README.txt file in the port/{port} directory, along (possibly) with CHANGES.txt and TODO.txt files.

9. Detailed Porting Guide

This section assumes the following basic approach. We'll be creating new chip ports following the two-step build strategy. Step 1 is to create the chip-specific libraries for the DataPump, and does not depend on the USB resource compiler or any particular target application. Step 2 is to create one or more test applications, using the resource compiler.

The DataPump has "sample" chip specific codes in the source tree. The sample chip codes reside at: {DataPump}/ifc/sample. These sample codes provide chip port framework. If you are using sample code, you can easily create new chip ports. At the end of this section, we will describe how we will create new chip ports from sample chip ports.

9.1 Step 1: Develop the DataPump Libraries

9.1.1 Choose names

- Choose a short name that will be used as the "official" name of the port. This should be file-system clean, as it will be a directory name, and all lower case.
- Choose a C-code prefix that will be used as the prefix of names specific to the port.
- Choose a filename prefix that will be used as the prefix for names of files specific to the port. For "C" files, we use a naming convention in which the first few characters are a (terse) chip-specific abbreviation. For example, "u820" is used for the USS820, "bcm" for the Broadcom 3310. Happily, the D12 is easy: "d12". The point of this is to make sure that the file names are likely to be globally unique, as that improves debugging in some environments.

9.1.2 Create the chip code location in the source tree

Normally, chip specific code resides at: {DataPump}/ifc/{chipname}/

The build procedure assumes the additional structure:

i/	include files
m4/	m4 header files
common/	the common code for that chip
packlist/	the release packing list used by "release-tools"

9.1.3 Develop the M4 file

This is the primary reference file that is used to access ports and registers, and for giving commands to the chip. The definitions are entered using M4, so we can easily generate assembly-language header files, too, if needed.

9.1.4 Create a UsbMakefile.inc file in ifc/*name*

If the port level files define the variable IFC=*name*, then the DataPump Makefiles will look for ifc/*name*/UsbMakefile.inc. Please refer to the UsbMakefile.inc from the uss820 or mpc850 ports for an idea of what this should look like. Make sure you include *chip*.h.m4 in the SOURCES list.

9.1.5 Develop the chipinfo.urc

Description and contents of this file is explained very well in the Engineering Report 950000061 (USBRC User's Guide)

9.1.6 Modify the make files and build procedures

We need at least to be able to create the chip register header file from the M4 file.

9.1.7 Determine the access methods

Normally we create at least macros for efficiently accessing the ports of the chip. For chips with extensive indirect addressing, we will also create macros that handle the indirect addressing step transparently – otherwise the code becomes impossible to read.

On some chips, such as the D12, it may be better to define specific subroutines for issuing commands. This is up to the architect – there are no hard-and-fast rules. Each hardware design requires a new approach for good performance.

9.1.8 Create the chip-specific instance-data header file

In order to do this, you must take an initial guess at the layout for the following structures.

1. UDEV*xxx*.
2. UEP*xxx*

In addition, you'll need to incorporate the access method macros.

9.1.9 Create the central chip files

This central file contains the UDEVICESWITCH instance for this chip. At this time, you'll need to assign names to the UDEVICESWITCH routines; and therefore you'll need to come up with a naming convention. Normally it will be something like "**uss820**_", "**bcm3310**_", or "**d12**_". These subroutines would be:

1. *xxx*_DeviceReportEvent
2. *xxx*_DeviceReply
3. *xxx*_DeviceStart
4. *xxx*_DeviceStop
5. *xxx*_DeviceRemoteWakeup
6. *xxx*_DeviceInitialize[1]

Current "best practice" is to put each of these routines into their own files. Suggested names would be:

- *xxx*devsw.c for the device switch.
- *xxx*devreply.c for the Reply function.
- *xxx*devreportevent.c for the ReportEvent function.
- *xxx*devstart.c for the Start function (which might not actually do anything – in most of our ports it does not).
- *xxx*devstop.c for the Stop function (again, it might not actually do anything).
- *xxx*devremotewakeup.c for the RemoteWakeup function.
- *xxx*devinit.c for the Initialization functions.

By the way, don't be misled by the fact that the USS820 has a file called "regtest.c" – this is not compliant with current practices.

Also, for consistency with the code base, the *xxx*_Initialize code might be defined to be identical to the original *xxx*PumpInit(). Or you might want to provide both an *xxx*PumpInit() for backwards compatibility, *and* the *xxx*_Initialize function. (If you provide an *xxx*PumpInit(), you should put it in a file of its own, so that it won't be linked if it's not needed.)

[1] New in version 2.0 of the DataPump.

9.1.10 Create the EpioInit routine needed by USBRC

USBRC creates the primary initialization code for the DataPump. In that code, it generates calls to a chip-specific subroutine that initializes the endpoint structures. The name of this subroutine is found in the chipinfo.urc file. It should be considered a primary interface to the DataPump.[2]

Since this is a primary interface, it deserves its own name and file. The recommended name is:

*xxx*_EpioInit

and the recommended file is:

*xxx*epio.c

Don't forget to change the name in the chipinfo.urc file!

9.1.11 Create the Endpoint Switches for the supported endpoint types

The DataPump needs to have instances of UENDPOINTSWITCH structures for each distinct kind of endpoint, meaning:

- Control OUT – *xxx*cnto.c
- Control IN – *xxx*cnti.c
- Bulk OUT – *xxx*blko.c
- Bulk IN – *xxx*blki.c
- Interrupt IN[3] – *xxx*inti.c
- ISO OUT – *xxx*isoo.c
- ISO IN – *xxx*isoi.c

Each such file will contain at least the UENDPOINTSWITCH. If there are multiple operating modes of the chip (for example to tune bulk transport for various usage scenarios), the porting engineer figure out how to accommodate these, perhaps by defining additional switches that can be used at runtime to override the basic definitions.

Frequently, it's possible to use the same subroutines, or even to use a common set of subroutines. It depends on the hardware. On the USS820, we use the same code for

[2] In future version, this routine should be moved to the UDEVICESWITCH, so that we can eliminate yet another "random" interface, and collect them all into the central structure.

[3] The version 1.5 data pump does not support Interrupt OUT, mostly because the USB Resource Compiler doesn't support it.

implementing Control OUT and Bulk OUT transport; and we use the same code for Control IN, Bulk IN, and Interrupt IN. On the BCM3310, because the hardware is much different for each endpoint, we have almost no common code.

A typical endpoint switch definition might be:

```
ROM UENDPOINTSWITCH gk_XXX_BulkOutSwitch =
        UENDPOINTSWITCH_INIT_V3(
                XXX_BulkOutStartIo,
                XXX_BulkOutCancelEpIo,
                XXX_EndpointBufPrep,
                XXX_BulkOutChannelEvent,
                XXX_BulkOutTimeout
                );
```

Notice that *XXX* is the device prefix (e.g., *uss820* or *bcm3310*). gk_ is the MCCI standard prefix for "global constant". The UENDPOINTSWITCH_INIT_V3() macro isolates your code from any future changes in the UENDPOINTSWITCH structure contents or layout.

The key thing to remember is that each device is different. We can study those routines for "the meaning of the UBUFQE flags", but not for "here's how to talk to the hardware". Each device has its own subtle ways of dealing with the bus.

The client indicates that they want us to move data by building a UBUFQE and submitting it to a pipe. For IN-bound packets, the client controls fine points of the data transfer by setting flags in the UBUFQE. The meaning of the flags changes depending on the type of the endpoint.

The key flags for IN UBUFQEs are:

- UBUFQEFLAG_SYNC -- attempt to defer completing the request until the host has collected the data (and ACKed). Hardware differs as to how well we can guarantee this, so this is a "best effort" flag.

- UBUFQEFLAG_PREBREAK -- make sure that a short packet is sent BEFORE the first byte of this buffer. I.e., if the previous UBUFQE didn't result in a short packet being sent (either implicitly or explicitly), then insert a ZLP (zero length packet) before sending the first byte of this buffer.

- UBUFQEFLAG_POSTBREAK -- indicates the end of a logical message. The client is asking the chip driver to make sure that a short packet will be sent as part of the last packet that is used to transport this buffer. There are two cases: either the last USB packet for this buffer is naturally short -- in that case, we don't need to do anything. Or else the last USB packet for this buffer is an exact multiple of the max packet size for the pipe. In that case, the chip driver must send a ZLP (zero length packet) after sending the last USB packet. The chip driver must queue the ZLP before completing the request.

These are the only flags that are important for device-to-host transmissions.

However, be aware that the client code assumes that the DataPump will coalesce packets that are submitted "all at once", if the PREBREAK and POSTBREAK flags aren't set. This means that if you have a max packet size of 64, and you get 10 messages of 12 bytes, you would send 2 packets: a 64-byte packet and a 120-64==56 byte packet.

Also be aware that the DataPump design assumes that the DataPump will not exit from put packet with a partial (uncompleted) packet in the device. The put packet code should finish the packet (by issuing the D12 "VALIDATE BUFFER" command -- weird name) before returning.

The PREBREAK flag means that the chip driver needs to keep track of short packets in each endpoint; each time a packet is sent, the flag must be updated. Even though this is a common problem, each chip driver has to provide this function, because intelligent hardware might do it internally.

Some hardware is not very nice about handling the FIFOs, particularly the USS820. On the D12, we can just fill, I think, and then VALIDATE BUFFER if we've put data in the FIFO but run out of data, just before we leave put packet.

On the D12, interrupt pipes can be handled identically to bulk pipes.

For OUT pipes, there are more flags:

- UBUFQEFLAG_SYNC has the same meaning but is irrelevant on the D12; the data is already there, and has been acknowledged. (It would mean, "don't tell the host that we've got the data until it's in our buffer", but that's not possible.)

- UBUFQEFLAG_STALL is used to stall an endpoint, create a packet with UBUFQEFLAG_STALL set, and with a zero-length buffer. When the packet reaches the head of the queue, the endpoint will be stalled.

- UBUFQEFLAG_SHORTCOMPLETES means that the chip driver should complete the UBUFQE when it copies a packet that is less than wMaxPacketSize into the buffer, even if the buffer is not full. If this flag is not set, then the chip driver should not use short packets as a completion criterion – the data should just be catenated into the buffer.

- UBUFQEFLAG_DEFINITE_LENGTH means that the client is expecting a message of specific length. It should be consulted by the chip driver in the following situation: there is more data in the packet in the FIFO than will fit in the client's buffer. If this flag is not set, then the chip driver should allow the last packet to be split across multiple UBUFQEs. If this flag is set, however, the chip driver should discard the extra data, and complete the UBUFQE with an error (USTAT_DEFINITE_LENGTH_OVERRUN).

- UBUFQEFLAG_IDLE_COMPLETES the caller should set this flag if the timeout value is to be used as an idle time.

- UBUFQEFLAG_PROTO_RESERVED is reserved for use by protocols.

In BulkOut Transfer also, the client always allocates UBUFQEs; the chip driver never allocates UBUFQEs. The client also allocates the buffers that the UBUFQE points to.

The allocated buffer size is allowed to be smaller than the incoming data. This is, however, an uncommon situation. The chip driver should optimize for OUT pipe buffers that are a multiple of the max packet size.

On the D12, it appears that we must always read a full packet from the device. (It also appears that CLEAR FEATURE (endpoint stall) clears the FIFO for OUT endpoints; but we'll deal with this complication later.) It also appears on the D12 that we can't find out how big the packet is without reading the first two bytes, and that this drains the FIFO.

So, each endpoint probably needs to have a slot to record how many bytes are left in the current packet. Normally, we'd read the count, and then copy the bytes. If there's not enough room in the current UBUFQE, we could look ahead in the pending UBUFQEs to see if there's enough room (that's what the 820 code does); but in this case, I think we should just set an endpoint flag (saying that there's partial data in the FIFO) and store the "remaining count" in the UEPD12, so we can recover it the next time we come through.

Discarding data in the case of things like UBUFQEFLAG_DEFINTE_LENGTH errors has to be done by reading and discarding the remaining bytes.

9.1.12 Create/modify a source release kit

The “release-tools” need information of specific chip ports to release source codes. A packlist file is the "release-tools" packing list for source-level releases of the DataPump chip driver for the specific chip. This file is read by release.sh to generate releases, by tag.sh to tag, and by checkout.sh to checkout. Recommended file name is source-release.pkl.

Also we should create special purpose chip ports release file that builds the .h files from the m4, and omits the m4 bits from the resulting file. Recommended file name is *xxx*-release.pkl.

9.2 Step 2: Develop the Test Applications

The loopback protocol is a generic, debugging & hardware verification protocol. The loopback simply echoes back whatever is sent to it across the USB from the host. This loopback protocol is included in the DataPump Firmware Development kit, and this protocol code resides at {DataPump}/proto/loopback. The loopback application can be used to test chip port. This section describes development procedure of loopback application. Please look at the {DataPump}/ifc/isp1362/apps/loopback for a good example.

9.2.1 Create the application location in the source tree

The test application code reside at: {DataPump}/ifc/{ifcname}/apps/{appname}. The location of loopback application of ISP1362 is {DataPump}/ifc/isp1362/apps/loopback.

9.2.2 Create a UsbMakefile.inc file in ifc/{ifcname}/apps/{appname}

UsbMakefile.inc is found by the DataPump build system, and is used when building the application "stand alone". UsbMakefile.inc contains the information about the common sources and library requirements for that application. It also contains the URC file and the table header that instantiates the application, to the list of required sources to be compiled and linked.

9.2.3 Develop the *xxx*.urc

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.

Description and contents of this file is explained very well in the Engineering Report 950000061 (USBRC User's Guide).

Once you have described the USB resource file, the USB resource compiler will generate two files:

- a C code (.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. This file also contains all the code needed to initialize the data structures at runtime to match the description given to the host.

- 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.

9.2.4 Create a simple initialization function

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.

9.3 Create Chip Ports Framework from Sample Chip Ports

The sample chip ports code reside at: {DataPump}/ifc/sample. The sample chip ports code contains following directories.

apps/	the sample test applications
apps/loopback	the loopback application

common/	the common code for sample chip
doc/	converting files from sample chip to specific chip
i/	include files
m4/	m4 header files
packlist/	the release packing list used by "release-tools"

9.3.1 Modify Sed Script File

The sed script file (named "new_chip.sed") is in {DataPump}/ifc/sample/doc directory. This sed script is for changing a skeleton file into an actual chip file. The porting engineer should copy this file to {DataPump}/ifc directory before modify actual chip information. This file contains following information:

```
# set the translation for "skl"
s/skl/xxx/g

# set the translation for "SKL"
s/SKL/XXX/g

# set the translation for "SAMPLE_CHIP"
s/SAMPLE_CHIP/Friendly-name-of-chip/g

# set the name of the .a file produced by the build
s/sample/xxxx/g

# set the translation for "Chip_Manufacturer"
s/Chip_Manufacturer/Who-Made-It/g

# set the date/time of creation and version strings
s/Mon Jan 13 2003 15:10:04/What-time-now/g

# set date for CHANGES.txt
s/2003-01-13/What-day-now/g

s/January 2003/xxxx yyyy/g

# set porting engineer
s/Terry Moore/Name of chip-level porting engineer/g
s/tmm/initials_porting_engineer/g

# set version number
s/1\.81a/Current_version/g

# change copyright date if needed
s/copyright (C) 2003/copyright (C) yyyy/g
```

The “skl” and “SKL” are same name that selected in 9.1.1 choose names section. The “SAMPLE_CHIP” is real chip name that is used in comments. The “sample” is used to create chip port location in the source tree like as {DataPump}/ifc/{chipname}/.

For example, create SEIKO EPSON S1R72005 chip ports framework. First copy ifc/sample/doc/new_chip.sed to ifc/s1r72005.sed file, and modify as below.

```
# set the translation for "skl"
s/skl/ r72005/g

# set the translation for "SKL"
s/SKL/ R72005/g

# set the translation for "SAMPLE_CHIP"
s/SAMPLE_CHIP/ S1R72005/g

# set the name of the .a file produced by the build
s/sample/ s1r72005/g

# set the translation for "Chip_Manufacturer"
s/Chip_Manufacturer/ SEIKO EPSON/g

# set the date/time of creation and version strings
s/Mon Jan 13 2003 15:10:04/ Wed Jan 29 2003 10:24:04/g

# set porting engineer
s/Terry Moore/ChaeHee Won/g
s/tmm/chwon/g
```

9.3.2 Generate New Chip Ports Framework

The sample chip ports has shell script file to generate new chip ports framework. This shell script (named "new_chip.sh") is in {DataPump}/ifc/sample/doc directory. The porting engineer should run this shell script in {DataPump}/ifc directory.

For example, to generate SEIKO EPSON S1R72005 port framework.

```
[d:/s/usb/usbkern]% cd ifc
[d:/s/usb/usbkern/ifc]% sample/doc/new_chip.sh s1r72005.sed
Creating s1r72005 directories
Creating s1r72005 files
Done.
```

The shell script, new_chip.sh created {DataPump}/ifc/s1r72005 directory. This directory contains followings:

UsbMakeifle.inc	Makefile for S1R72005
apps/loopback/	Sample loopback application for S1R72005
loopback.c	Define application init vector. Don't modify.
UsbMakefile.inc	Makefile for loopback application. Don't modify.
r72005loop.urc	USB resource file. Need to modify.

common/	Common code for S1R72005
chipinfo.urc	USB resource file for S1R72005.
r72005blki.c	Bulk in endpoint code
r72005blko.c	Bulk out endpoint code
r72005cnti.c	Control in endpoint code
r72005cnto.c	Control out endpoint code
r72005devinit.c	Device Initialize function
r72005devio.c	Cancel all device I/O
r72005devremotewakeup.c	Device RemoteWakeup function
r72005devreply.c	Device Reply function
r72005devreportevent.c	Device ReportEvent function
r72005devstart.c	Device Start function
r72005devstop.c	Device Stop function
r72005devsw.c	Device switch
r72005dmmy.c	Endpoint switch for invalid endpoints
r72005epfn.c	Endpoint functions
r72005epio.c	Endpoint I/O function
r72005inti.c	Interrupt in endpoint code
r72005isoi.c	Isochronous in endpoint code
r72005isoo.c	Isochronous out endpoint code
r72005isr.c	Initialize interrupt handling
r72005prisr.c	Primary ISR
r72005probe.c	Probe chip
r72005reset.c	Reset handling function
r72005resume.c	Resume handling function

r72005scisr.c	Secondary ISR
r72005suspend.c	Suspend handling function
r72005trace.c	I/O trace functions

i/ Header files

r72005kern.h	S1R72005 include file

m4/ M4 files

BSDmakefile.release	Release generator for S1R72005 chip include file
r72005.h.m4	.h header file generator
r72005.m4	S1R72005 register description file

packlist/ Release-tools packing list files

s1r72005-release.pkl	Special-purpose S1R72005 release file
source-release.pkl	Packing list for source release of the S1R72005 chip

The porting engineer should port marked "XXX" in generated files. For example, in generated ifc/s1r72005/common/chipinfo.urc file, there are two places marked "XXX".

```
# XXX Porting:  fix this to match hardware:
Number-of-Endpoints          8

% The only requirement for the mapping is that, %
% for each endpoint address, there is only one  %
% corresponding endpoint index                  %
Endpoint-Mapping
      {
      # XXX Porting:  fix this table
      ( 0,       0)
      ( 0x80,    1)
      ( 1,       2)
      ( 0x81,    3)
      ( 2,       4)
      ( 0x82,    5)
      ( 3,       6)
      ( 0x83,    7)
      }
```

The S1R72005 has 6 endpoints: endpoint 0 (control IN and OUT) and 5 endpoints (endpoint a through e), which can be individually defined as interrupt / bulk / isochronous, IN or OUT endpoint. The endpoint 0 is used for control transfer only, and endpoint 0 is used for IN and OUT transaction by setting the transfer direction. Then the endpoint 0 is physically one endpoint, but this endpoint looks like two endpoints. The ifc/s1r72005/common/chipinfo.urc file should be modified as below.

```
# XXX Porting:  fix this to match hardware:
Number-of-Endpoints           7

% The only requirement for the mapping is that, %
% for each endpoint address, there is only one  %
% corresponding endpoint index                  %
Endpoint-Mapping
      {
      # Note: S1R72005 support 1 control endpoint, 5 other # endpoint.
      Number of physical hardware endpoint is 6. # Please check
      maxnumber of active endpoint at the
      # same time.
      #  index endpoint direction type
      #  ----- -------- --------- ---------------------------
      {  0,    0,       out       control }
      {  1,    0,       in        control }
      {  2,    1,       out in    bulk interrupt isochronous }
      {  3,    2,       out in    bulk interrupt isochronous }
      {  4,    3,       out in    bulk interrupt isochronous }
      {  5,    4,       out in    bulk interrupt isochronous }
      {  6,    5,       out in    bulk interrupt isochronous }
      }
```