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 V3.0 Platform Porting Guide

Engineering Report 950000260
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	2004/07/09	Original release
Rev. B	2011/06/23	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

1. Introduction

This document describes the roadmap for creating a concrete platform code base by starting with the sample MCCI platform code. The document should be used by porting engineers for creating customer specific platforms.

1.1 Glossary

to block — the standard RTOS concept of delaying a task until a condition is satisfied, by marking it as not ready and scheduling other, possibly lower priority tasks. "Block" is used instead of "wait", because "wait" is often used to indicate busy-waits, where the CPU repeatedly tests for a specific condition, or timed waits, where the CPU delays a given task until a specific time interval has elapsed.

2. Roadmap

Here's what we plan to do:

1. Create a concrete operating system lower edge from the template operating system
2. Create a concrete platform initialization subroutine from the sample code for the port
3. Modify the MCCI USB DataPump® build procedure to build libraries for the concrete platform
4. Modify the target platform's init code to call our init code
5. Modify the target platform Makefiles to link with our code
6. Build, download, and test

3. Creating a Concrete Operating System

Operating systems shipped by MCCI are stored at `c:\mcci\DataPump\os\osname`. This "osname" is important, because it will be used in many places as the name of operating system. In this example, we'll use "linux" as the operating-system name.

We will also use the "sample operating system" kit. This kit is not part of the 3.0 CD, but is available from MCCI.

3.1 Directory and file naming changes

We do the following:

- Copy os/sample/doc/new_os.sed to os/sample/doc/new_linux.sed
- Edit new_linux.sed to change the various fields like date/time of creation, name of chip-level porting engineer, change version if needed, and change copyright date if needed appopriately.
- cd DataPump/usbkern/os
- Execute the below command
    - sample\doc\new_os sample\doc\new_linux.sed linux

Now the contents of os\linux should be:

```
[<drive>/usbkern/os/linux]% dir – R

OsConfig.inc            OsSetup.inc             UsbMakefile.inc

Directory of common:
ulinuxalloc.c           ulinuxdbgprint.c        ulinuxgetev.c
ulinuxpost.c            ulinuxalloctrack.c      ulinuxdi.c
ulinuxinit.c            ulinuxsetpsw.c          ulinuxchkev.c
ulinuxfill.c            ulinuxintr.c            ulinuxswc.c
ulinuxcpybuf.c          ulinuxfree.c            ulinuxioctl.c
usbpump_linux_internals.h

Directory of i:
usbpump_linux.h              usbpump_linux_debug.h
usbpump_linux_alloctrack.h   usbpump_linux_intr.h

Directory of mk:
buildset.var

Directory of packlist:
source-release.pkl

[<drive>/usbkern/os/linux]%
```

3.2 Lower edge interface functions

3.2.1 upf_pMalloc

This is converted to a call to the lower-level memory allocator, if such is available.

The DataPump core only calls this routine at startup time, so it's not really necessary for the base operating system to support dynamic memory allocation, but we normally use the OS's memory allocator if one is available.

3.2.2 upf_pFree

See malloc(). Some operating systems don't record the size of memory blocks in the pool, and need you to give them the size of the block when free()-ing. Therefore, the DataPump always passes in the allocated size. For many operating systems, this information can be ignored.

3.2.3 UHIL_cpybuf, UHIL_fill

These are simply wrappers for memcpy() and memset(), if the ANSI routines are available. Otherwise we must supply routines that implement memcpy()-like and memset()-like functions. See the function headers. The sample routines assume that memset() and memcpy() are available.

3.2.4 UHIL_di, UHIL_setpsw

These are routines that respectively disable and restore CPU interrupts. On some operating systems, these routines map directly to primitives supplied by the OS (spl() and splx(), for example). On others, we must use a static variable to track the "prior state".

Note: if we use a static variable, then we must also take extra care during the interrupt service routines to properly match the "simulated current state" to the actual state of the kernel.

3.2.5 UHIL_swc

This routine handles "software checks", which are failures that are detected by the software. (The word "check" in "software check" comes from very old IBM terminology; old IBM mainframes would stop with "machine check" failures or "hardware check" failures for parity errors or other hardware problems. This was generalized in the Xerox Sigma 7 operating system to include self-detected software failures. So a "software check" is essentially an assertion failure.

When the DataPump calls UHIL_swc(), it means that the DataPump is critically misconfigured, miscompiled, or otherwise totally non-functional. There's no way for the DataPump to safely continue. So it calls this routine, passing in an error code.

The OS-specific implementation should use the OS's assertion-failure mechanism if one is available. Otherwise, it should log a failure, and then enter a `while(TRUE)`-loop that does nothing.

3.2.6 upf_pPostEvent, GetEvent, CheckEvent, and the event queue

These three routines support the implementation of the DataPump messaging system. An "event" is a pointer to a __TMS_CALLBACKCOMPLETION structure. These structures are

statically allocated and managed by the DataPump. Each such structure contains two fields: a pointer to a function, and a context pointer for use by that function. To "post an event," the DataPump code initializes a CALLBACKCOMPLETION object and passes a pointer to the PostEvent code. Later, the DataPump main event loop dequeues that pointer, and calls the specified function. The DataPump is explicitly allowed to queue a CALLBACKCOMPLETION multiple times, even before the event gets dequeued.

Therefore, the semantics of upf_pPostEvent should be to put a **pointer** to the CALLBACKCOMPLETION into a queue. The queue can be implemented as a fixed-size ring buffer, or as an operating system queue. Depending on your OS, a fixed-size ring buffer may be much more efficient.

However, you also have to consider GetEvent. GetEvent allows a task to wait if the queue is empty. If you implement a ring buffer, you also have to implement some (OS-specific) means of blocking the task until the buffer is not empty. Often it is so hard to implement this, that it is easier to use the OS-supplied queue mechanism, at least for your first implementation.

On systems that implement OSE-like "signals", we normally assign a single "signal code" that is associated with PostEvent. Then GetEvent uses that signal code.

The DataPump needs an API that will look at the event queue to see whether a message is waiting. However, that API is NOT supposed to remove the event from the queue. So the implementation of CheckEvent should check the event queue without removing the event from the queue. Some operating systems provide a non-blocking form of the receive command, but always remove the first event if one is pending. This is a problem, because the DataPump "check for event" API does **not** have any way to return this event. Therefore, on these operating systems, we have to use the OS "non-blocking receive", and then save the first received event internally, so that GetEvent will return it later. (Sometimes operating systems use the words, "this API will not cause a calling task to wait" to mean "this API is non-blocking".

3.3 Interrupt Handling

3.3.1 Background Information

In order to be very portable, MCCI has a very abstract interrupt subsystem. Sometimes mapping the abstraction onto the platform's capabilities is easy; sometimes it is hard.

We will start by describing how the ISR gets registered and called.

MCCI's code views an ISR as a *closure,* in the sense that an ISR is both a function and a context pointer. The function is defined at compile time. The context pointer is computed, and points to a dynamically allocated structure. Therefore, when MCCI registers an ISR, it expects to call an API that records both the function pointer and the context pointer. Later, each time the interrupt occurs, the function will be called using:

```
(*pfn)(pContext);
```

The term "context pointer" means that this pointer gives the function the ability to find the actual dynamically allocated structure, and it makes the ISR function completely re-entrant. This "context pointer" is not interpreted by the interrupt subsystem; the interrupt subsystem just saves it when the DataPump registers the interrupt, and then passes that saved value to the ISR.

pContext is therefore a `VOID*`.

So the full prototype of an MCCI ISR is:

```
VOID
UHIL_INTERRUPT_SERVICE_ROUTINE_FN(
      VOID *pContext
      );
```

In fact, we have a typedef that defines `UHIL_INTERRUPT_SERVICE_ROUTINE_FN` and `PUHIL_INTERRUPT_SERVICE_ROUTINE_FN` in `i/uhilintsys.h`.

So the DataPump prototypes its ISR functions in the chip-driver header files by writing:

```
UHIL_INTERRUPT_SERVICE_ROUTINE_FN MyIsrFunction;
```

It declares the implementation as normal:

```
VOID
MyIsrFunction(
      VOID *pContext
      )
      {
      // body of ISR
      }
```

The DataPump is designed to be completely reentrant, and does not use global variables. However, the interrupt subsystem will often need some storage in order to associate a given ISR and context pointer with a specific hardware interrupt. Therefore, additional API values are defined to allow the interrupt subsystem to also be completely reentrant, if the implementer decides to make it reentrant. ***The DataPump does not require that the Interrupt System be re-entrant. But the design of the DataPump allows the Interrupt System to be re-entrant, if you like.***

This complicates the API.

Each call to the interrupt system implementation requires that some kind of reference pointer be passed in.

The full sequence of events for registering for an interrupt is:

1. Opening an interrupt connection handle

2. Connecting an ISR to the interrupt specified by the connection handle

3. Enabling the hardware interrupt specified by the connection handle.

Since some hardware has multiple interrupt sources, opening a connection handle requires two kinds of information:

1. The main interrupt system interface pointer

2. An interrupt resource handle that identifies the interrupt.

3.3.2 Interrupt Resource Handles

The Interrupt Resource Handles are very simple in practice, but very hard to describe because they are so abstract. They are 32-bit integers. The DataPump doesn't have any idea what these integers mean; so they are treated as *handles:* arbitrary numbers that are not examined. If two Interrupt Resource Handles are numerically identical, then they have the same meaning, but otherwise nothing is known about them by the DataPump.

The meaning of Interrupt Resource Handles is defined by the interrupt system implementation. Normally, each interrupt system implementation will define its own encoding scheme. The kind of information that might be encoded includes the vector number, whether it's a normal IRQ or an "FIRQ" (fast IRQ)[1]. The important thing is that all the important variations must be represented in this encoding, so that the DataPump just needs to pass in this handle, and the Interrupt System implementation will then know what to do. If the Interrupt System Implementation only supports a single interrupt, and that interrupt can never be changed, then only one Interrupt Resource Handle value need be defined, and the interrupt system can ignore the values actually used. But because of API requirements, the handle will always be passed in.

By convention, an Interrupt Resource Handle that is 0 is the "NULL" interrupt resource handle. This means that the interrupt system implementation should not use 0 to represent an actual interrupt. On systems where the natural representation would be a number in the range (0..n-1), where the number represents a interrupt vector number, MCCI's convention is to use the value 1 to represent vector #0, 2 to represent vector #1, etc. These handles are very infrequently used; they are only used when opening a connection handle, so the computational cost is not significant.

DataPump chip drivers need to register for interrupts. They get their Interrupt Resource Handles from their "platform wiring table". This table defines how the USB section is wired up on this particular hardware implementation. In the source code for that table, the porting engineer puts the appropriate Interrupt Resource Handle in the "platform wiring table" entries. During DataPump initialization, the chip driver then uses those handle(s) to register for interrupts. So all the information about how to wire up interrupts is outside of the MCCI core code, and you don't have to edit anything in the chip driver to change how interrupts are handled.

[1] In the Arm and some other architectures, "Fast IRQs" are higher priority than normal IRQs, and save less context when the hardware processes the interrupt. Depending on how the base software works, an FIRQ might be similar to an "NMI" (non-maskable interrupt), because the base software might be designed never to mask FIRQs. The Arm has separate mask bits for FIRQs and normal IRQs.

For example, the Interrupt Resource Handle on MCCI's os/none, which uses the Analog Devices (ADI) core and ARM processor, encodes the following information:

1. The interrupt number, from 0 to 15 – this is as defined by the ADI core

2. The interrupt group, also from 0 to 15 – this is defined by the ADI core

3. A flag indicating whether the ISR should be run in Thumb mode or ARM native mode.[2]

If you are not using os/none, you have to study how your platform handles interrupt connections. If your platform is also AMR based in it requires that all ISRs have to run in ARM mode, and if you will be running the DataPump in ARM mode, then this flag is not needed.

If your platform code requires that all interrupts be manually configured, then the Interrupt Resource Handle might not carry any meaning. But you still need to give the DataPump the interrupt APIs it needs.

3.3.3 Planning the Interrupt System

You must first find out how your platform handles other interrupts.

You must consider:

- How does the CPU handle the interrupts at the lowest level? This information is usually found in the CPU architecture manual.

- How is the interrupt controller used? Normally, the chip vendor incorporates an interrupt controller, but this will differ from chip to chip, even if the CPU type is the same. Some software is normally needed to read the interrupt number from the interrupt controller and decide which interrupt is pending.

  Because the USB system must be integrated with all the existing code, it is critical to first understand how to add interrupt logic to the existing code.

- Finally, the platform may have high-level (portable) configuration options.

Our strategy for a simple OS without dynamic interrupt configuration will be:

1. Insert the code to call an interrupt subsystem function from the lowest level code into the lowest level code, if needed. (If the low-level code is written in a general-purpose way, this might be done via a function call. Otherwise, it might be done using macros. For example, you might have to add something like INT_DEF() to a table, and then add functions that somehow create the linkage.

[2] If the ISR is compiled in Thumb mode, this flag should be set. If the ISR is compiled in ARM mode, this flag should be clear. The low-level ISR code on an ARM always starts in ARM mode, but can switch to Thumb mode if it likes. If the body of the DataPump is compiled in Thumb mode, it's very convenient to keep the DataPump ISR also in Thumb mode, as it keeps all the compilations consistent.

2. The DataPump interrupt subsystem function will use a static or global variable to find the pointer to the interrupt function, and to find the saved context pointer. It will then call the registered function.

3. The interrupt subsystem will support exactly N interrupts; the configuration of which interrupt is used will be done by the platform layer configuration. N will be defined at compile time. The low-level routines will be named DataPumpIsr1, DataPumpIsr2, etc.

4. So as an example the actual implementation of the interrupt layer for the DataPump for the ADI will be quite simple.

3.3.4 Data Structures

For convenience, we define a parameter, DP_NUM_ISRS. The value of this constant determines how many ISRs can be supported. We then use #if's to select between 1 and 8 ISRs in the conditional compile.

We separate the pure code from the conditionally compiled components. The conditionally compiled components need to be placed in the central platform directory. We therefore have three files:

`os/linux/common/ulinuxintr.c` – has the pure code

`os/linux/common/ulinux_param.c` – has the code that has to be compiled in the linux environment, and which changes based on the platform configuration.

`os/linux/i/dp_linuxintr.h` – the header file that defines the linkage between the two parts.

Since we don't want to recompile the common code each time, we put the value of `DP_NUM_ISRS` into a global, `gk_ulinux_DataPumpNumIsrs`, for reference by the common code.

Interrupt connection handles are defined as pointers into a global array of interrupt state blocks, of type `DPISR_STATE`. Each element contains management info, plus a pointer to a const array of information about each interrupt. The main purpose of the const array of information is to allow us to enable and disable each interrupt separately without having to include the linux configuration files into the common code.

The overall interrupt subsystem object is defined as a const global by `os/linux/common/ulinuxintr.c`, with the name `gk_ulinux_interrupt_system.` This name is only used by the initialization code; after that, it is referenced through a pointer in the UPLATFORM structure.

The remaining code is almost self-explanatory. OpenInterruptConnection returns the appropriate pointer into the interrupt state table, and marks the entry as busy. CloseInterruptConnection releases an open handle. ConnectToInterrupt stores the incoming ISR pointer and the context pointer in the state table entry. SetInterruptState uses the

information table to locate the low-level interrupt enable/disable code, and calls the appropriate routine.

We require that the platform configuration files define one or more interrupts for the DataPump. The macro for use by `M_EnableHwInt()` must be named `DataPumpUsbInt0`, `DataPumpUsbInt1`, etc. This macro must in turn define the ISR as being `DataPump_LowLevelIsr_0`, `DataPump_LowLevelIsr_1`, etc.

3.4 Debugging Interface

When compiled for debugging, the DataPump generates text messages to an abstract output stream. If you want to be able to see these messages, you need to write code that connects this abstract output stream to the appropriate concrete mechanisms in your platform.

Most platforms already have some kind of debug mechanism defined. On Windows and MCCI's TrueTask research OS, there is a kernel error log. On other systems a debug UART is used.

The DataPump uses an object-oriented API for logging, so that the implementation can be adjusted for the platform. The object is created by calling a platform specific function by name; this function is normally named uXXX_DebugPrintInit(). This function returns a debug object. Embedded in the object are several function pointers, which define an API for use by the DataPump. This API has the following functions:

pCloseFn	function to be called to close the debug system
pPrintBuf	a function that prints a buffer, converting \n to \r\n if necessary. This function must not block. If there is no room for the debug data, the data must be discarded.
pPrintBufTransp	a function that prints a buffer without any interpretation (if possible). Again, this function must not block. If there is no room for the debug data, the data must be discarded.
pFlush	a function that causes the caller to wait until all debug information has been transferred to the logging device. This function may block, but it is a good idea to have a timeout.
pPrintPoll	an optional function. If non-NULL, the DataPump event loop will call this function periodically. This is used (for example) in os/none, to send characters to a non-interrupt-driven UART.
pDebugPrintEnable	a function that turns debug logging on or off dynamically.

The most important thing to consider is the effect logging will have on performance. If the platform has a high-performance logging function "built in", then pPrintBuf and pPrintBufTransp can simply copy the data to the logging function.

If the platform logging function is documented as "blocking" or "might slow down the system", then an intermediate logging thread should be created. The os/nucleus port has a good example of how to do this.

It's easier to write the debugging task if the OS supports event flags, but it's possible to use a signal queue to notify the debugging task that it's time to poll the buffer.

3.4.1 Making the debug code go away

If you do not want debugging, you should compile the library and initialization with the "free" option (or link with the free libraries). If you have a static operating system, and you are using a debug thread, you will also want to disable the debug thread; or else provide a stub version that does nothing.

You want to be careful not to link in the full debug code when you are not debugging, because it will pull in a lot of code from the library. This is one reason that we put the debug code in a separate module with a separate API for accessing it; if you don't reference the "open" API, the code will not be linked into your image.

3.4.2 Platform Level Issues

The MCCI sample OS has one additional feature. There is the portable UPLATFORM, and this is contained in the UPLATFORM_*MYOS* structure. The UPLATFORM_*MYOS* contains instance data for the DataPump that is specific to the local operating system. Of course, it can be different from operating system to operating system.

One of the fields in the sample UPLATFORM_*MYOS* structure is a pointer to a function that is to be used for debug output. The prototype for this function can be changed as needed by the porting engineer, because it is only used in code that is in the os/*myos* directory. In the sample code, it has the prototype VOID (*pDebugPutString)(CHAR *).[3]

The sample debug code copies this function pointer from the UPLATFORM_*MYOS* structure into the debug context structure (U*MYOS*_DEBUG_CONTEXT), and later uses this function for outputting characters.

The porting engineer should consider whether this is appropriate, or even needed.

The sample OS uses this function to actually put the characters to the native logging function.

There are three approaches to porting:

[3] Please note that this prototype is not really correct; it should be VOID (*pPrintDebugString)(CONST CHAR *). But the sample OS is based on the Nucleus port, and the Nucleus low-level debug output routine uses CHAR * rather than CONST CHAR *.

1. Simply call the operating system's debug output routine directly. Unless you are making code that will become part of an MCCI standard product, this is probably acceptable.

2. Create a function that matches the prototype of pDebugPutString, and writes to the operating system

3. Change the prototype to match whatever your OS needs.

If you use the indirect function call approach, it makes it somewhat easier to handle the "debug"/"non-debug" version distinction. But it also makes the code harder to understand.

No matter how you do this when working with the sample code, *this* pDebugPutString can block as needed. The portable DataPump logging system never uses it directly.

3.5 Initialization

Initialization in the sample OS code is somewhat complicated, because it's based on MCCI's needs for common code that will address every customer's needs. If you are porting for your own purposes, then you may not need to write such a complicated system.

However, if you are porting the sample OS code, it may be quicker just to use the structures defined, and edit them for your needs.

Remember, though, that the initialization code is defined by you, and called by you, so you have a lot of flexibility as long as you meet the DataPump's requirements.

3.5.1 The Initialization Framework

For maximum flexibility, the MCCI sample initialization function centralizes all possible initialization functions. It gets its configuration information from a table that is passed in. The name and layout of this table are OS-specific. For reference, the key structures and functions used are:

u*myos*_UsbPumpInit()	a function, normally returning BOOL, that does all the DataPump initialization. See section 3.5.2
UPLATFORM_*MYOS*	a RAM-based structure that represents the operating system to the DataPump common code. See section 3.5.3
U*MYOS*_CONFIGURATION_-INFO	a CONST structure (normally in ROM) that represents application-specific configuration options to u*myos*_UsbPumpInit(). On operating systems that allow dynamic configuration, the information in this table might include task priority, stack sizes, and other tunable parameters.

By using this framework and adjusting it to your requirements, your initialization code can be written without any conditional compiles. This normally leads to much better reliability. However, it may also be more trouble than it is worth!

3.5.2 The Initialization Function

The sample code initialization function has the following prototype:

```
BOOL
umyos_UsbPumpInit(
      UPLATFORM_MYOS                                    *pPlatformMyOs,
      CONST UMYOS_CONFIGURATION_INFO                    *pMyOsConfig
      VOID                                              *pPortContext,
      CONST USB_DATAPUMP_PORT_INIT_VECTOR_HDR           *pPortInitVecHdr,
      CONST USB_DATAPUMP_APPLICATION_INIT_VECTOR_HDR    *pAppInitVecHdr,
      UDEVICE                                           **pvpDevices,
      ULONG                                             nDevices,
      UINT32                                            ulDebugMask
      )
```

The sample OS code requires that the *caller* allocate the memory for the UPLATFORM_*MYOS* structure. This allows the caller to decide whether to put the platform in memory, if desired. You can change the API to use the native OS allocator, or even to use a global variable, if you want. In that case, you probably should change u*myos*_UsbPumpInit() to return a pointer to the UPLATFORM_*MYOS*, or NULL if it can't allocate one. However, some RTOS kernels don't have a native allocator. (For example, some require that you specify the pool to be used, and there is no default.) Therefore, the sample OS code requires that the caller take care of allocating the object.

The `pPortContext` pointer is not interpreted by the Sample OS layer, or by the DataPump. It's simply put into the upf_pContext field of the portable UPLATFORM structure. It can be NULL (or omitted from the API) if you don't need it.

pPortInitVecHdr is the interface to the DataPump code that looks for USB device hardware. Often, you only have one possible USB device hardware setup, so this might not be needed. However, it's good practice to use this, and it lets you quickly test with other USB hardware if needed. It also makes your code compatible with the MCCI test applications.

pAppInitVecHdr is the interface to the DataPump code that binds the USB class protocols to the device described by your descriptors. This is always needed, unless you need to write your own application initialization logic. If you are going to use MCCI sample applications for test purposes, you will need this parameter in your initialization function.

`pvpDevices` is a pointer to an uninitialized array of pointers to UDEVICEs. It will be filled in by the initialization logic based on the devices that are actually found in the system. `nDevices` tells the initialization code how many slots are in the array. After the DataPump finds that many physical USB device ports, the DataPump will stop looking.

`pMyOsConfig` points to a read-only structure of type U*MYOS*_CONFIGURATION_INFO. Depending on your OS, this structure might not be needed. See section 3.5.4.

`ulDebugMask` sets the initial value of the debug mask, using bits defined in i/usbpumpdebug.h. The flags defined are:

```
UDMASK_ERRORS       (1L << 0)       trace gross errors
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
```

A good starting value for debugging is 0x0B (CHAP9 | ANY | ERRORS).

The reason ulDebugMask is in the API is that you might want to change this at startup based on a value saved in NVRAM.

The DataPump reference manual documents the DataPump startup requirements. To summarize, the DataPump requires the following:

1. First call UHIL_InitVars()

2. Set up anything needed in the UPLATFORM_*MYOS* structure (apart from the core DataPump values.

3. Call UPLATFORM_INIT_V1() to set up the DataPump-required values.

4. Find the DataPump event dispatching context block, and set uevent_active = TRUE. This tells the APIs that an active thread is used for dequeuing the DataPump events. (If left at false, the DataPump event logic will run based on "opportunistic" scheduling, by stealing the current thread as needed. If you don't have a scheduler, this is how you have to set things up. But usually, you'll have a scheduler; in this case opportunistic scheduling is frowned upon because it doesn't present the USB function as a separate task, and it defeats prioritization).

5. Create, and if necessary activate, the USB task.

The Sample OS USB task completes initialization by doing the following:

1. Calling UsbPump_GenericApplicationInit() to probe all the hardware and set up all the device class protocols

2. Printing a debug message showing the device information

3. Calling UDEVSTART() for each device that was found in step 1.

4. Entering a loop that waits for signals and then dispatches them as needed.

On a static operating system, you may want to rearrange the logic. The same steps must be performed, but normally the USB task is started automatically at boot time. This means that it is probably convenient to put all the USB initialization code into the USB task.

However, this means that either the "main" routine for the USB task must be external to the OS-specific library, or else the user will have no way to use the configuration disciplines given above.

We normally chose to put the main routine external to the USB task, and have two functions:

1. u*myos*_UsbPumpInit, which performs all of the steps EXCEPT for step 4 of the second procedure.

2. u*myos*_UsbPumpRun, which encapsulates step 4 of the second procedure.

3.5.3 The UPLATFORM_*MYOS* structure

This structure must be defined for your operating system. You must put in any information needed by the files in os/*myos*/common. You must also write the code to initialize it, as part of the call to u*myos*_UsbPumpInit().

It is a good idea to delete the fields that you do not need on your OS and with your implementation.

In the sample OS, some of the fields are used to carry information from the initialization code (which runs as part of the boot task) to the USB Task (which is created by initialization logic). These fields are not needed in static operating systems.

The information in UPLATFORM_*MYOS* includes:

u*myos*_Platform	The actual UPLATFORM structure, as the first entry in the larger structure.
u*myos*_Config	The configuration information gets copied into the platform structure
umyos_pAbstractPool	Internal memory pool used by the DataPump for dynamic allocation of memory.
u*myos*_EventContext	A standard USB DataPump UEVENTCONTEXT object. This is used for event management.
umyos_StateFlags	The platform state flags

u*myos*_pPortVecHdr	Port vector header
u*myos*_pAppVecHdr	Application vector header
u*myos*_pvpDevices	A pointer to the vector of devices found during initialization
u*myos*_nDevices	The number of slots in the vector. This is NOT the number of devices actually found!
u*myos*_InitialDebugMask	The debugging mask

3.5.4 The U*MYOS*_CONFIGURATION_INFO structure

If used, this structure will contain information used for tuning the USB DataPump's use of base operating system resources. In the Sample OS code, this information includes:

nEventQueue	DataPump Event Queue information
pMemoryPool	Platform memory pool structure to be used for dynamic memory allocation by the DataPump
nMemoryPool	Number of bytes of memory
pDebugPutString	Debug print string function
pInterruptSystem	Platform provided interrupt system interface
pTimerSwitch	System timer switch
pAllocationTracking	Dynamic memory allocation tracking system
pIoctlFn	Platform IOCTL related function

3.6 Header files

The os-specific header files for the DataPump are in os/*myos*/i.

3.6.1 dp_*myos*.h

This header file should be included by code in your operating system that needs to call code from the DataPump operating system layer. It provides definitions for UPLATFORM_*MYOS*, and for the function u*myos*_UsbPumpInit().

If you have a static operating system, it should also include a definition for u*myos*_UsbPumpRun(). Your USB task should call UsbPumpInit(), and then UsbPumpRun().

The name is conventional; you can call it whatever you like. Apart from your code, this header file is only used by modules in the os/*myos*/common directory.

3.6.2 dp_*myos*i.h

This header file contains internal definitions for use by the implementation of the operating system layer.

The name is conventional; you can call it whatever you like. Apart from your code, this header file is only used by modules in the os/*myos*/common directory.

Any internal data structures and functions to be called across modules within the OS implementation should be defined here.

3.6.3 dp_*myos*dbg.h

This header file contains definitions specifically for the debugging module. It is only used in u*myos*dbgprint.c. It includes dp_*myos*i.h and uhildbg.h, and defines the platform's debugging context object. It is in a separate include file mainly to allow you to divide u*myos*dbgprint.c into several files if you need to.

3.7 Makefiles

Three makefile fragments are created at the top of the os/*myos* directory. These are included by various portions of the DataPump make system.

`os/myos/OsSetup.inc`

This makefile simply defines any additional variables that are needed for compiling code that is aware of os/*myos*. In the sample OS, the makefile defines the variable EXTRA_CPPINCPATH to point to the directories found in the variable CCINCPATH_OS_linux_extra. This variable is initialized on OsConfig.inc to point to os/*myos*/i. If you need additional fields, you can add them here.

Note that OsSetup.inc does *not* call OsConfig.inc. This is intentional. OsConfig.inc loads the required context for an operating system into the BSDmake environment; OsSetup.inc makes the context active.

`os/myos/UsbMakefile.inc`

This makefile is used when building the operating-system library for the DataPump. It sets the variable SOURCES to a list of the files in os/*myos*/common directory that are to be compiled.

This include file is automatically included by DataPump, based on the value of the OS variable (which should be set to *myos* in your buildset.var file, or in your Config.mk file, created by

makebuildtree). It is called from os/UsbMakefile.Inc, which is in turn called from mk/libdesc.mk

In order to support arch- and port-specific code for your operating system, you should call ${SRCDIR}/os/OsMakefile.inc at the end of this file. That makefile simply searches through the OS arch/ and port/ specific directories, looking for additional UsbMakefile.inc files, which can provide additional source files to be compiled. These files need not exist, so it is always safe to call ${SRCDIR}/os/OsMakefile.inc at the end of os/*myos*/UsbMakefile.inc.

`os/myos/OsConfig.inc`

This makefile is included by other makefiles, to set up variables that can be used in compiling applications for *myos*. Conventionally, this header file assumes that the environment variable `MYOS_HOME` is set, and points to the top of the operating system tree. This header file will set `MYOS_INCPATH` to point to the standard header file directories in the operating system directory; the directories are listed separated by spaces. If `MYOS_INCPATH` is already in the environment, then it will be used; otherwise, it will be computed based on the value of `MYOS_HOME`.

If the operating system has "variants" (for example, different versions or different configurations), then this makefile calls osvariant.mk from the DataPump mk directory. This allows you to use OS variant values in makebuildtree and in your buildset.var. If there are no variants in your operating system, you can delete this part of the makefile.

This makefile should also define any other variables that may be needed or referenced from other makefiles. Important variables are:

`BUILD_APPS_AS_LIBRARIES` – if 1, then the DataPump build procedure will build applications as .a files. This is useful for test applications, and when building as part of a larger static embedded operatings system.

`CCINCPATH_OS_myos` – set to the include paths needed for your operating system. Used by the main makefiles.

`CCINCPATH_OS_myos_extra` – set to the include path to the os/*myos* include files, i.e. to os/*myos*/i. If you have a very complicated implementation with OS header files in several subdirectories of os/*myos*, then please add those directories to this variable.

*MYOS*_ARCH – normally set to the DataPump arch-specific

directory. Sometimes the OS has its own rules for arch-specific directories.

`MYOS_PORT` – normally set to the DataPump directory that contains the code for your specific platform. Sometimes the OS has its own rules for port-specific directories.

`MYOS_ENV` – normally set to the DataPump `BUILDTYPE` variable, which specifies whether this is a checked or free build. For some operating systems, the OS uses a different naming convention which must be followed in setting this variable.

The `OsConfig.inc` file *must* define the target `__OsConfig__`: as a dummy target. It should use this to avoid multiple includes of the file, using the following mechanism:

```
.if !target(__OsConfig__)
__OsConfig__:

# actual Makefile code goes here.

.endif
```

3.8 Buildset.var

With this setup, we can try compiling.

The easiest first step is to find a port that is close.

Find arch/*/port/*/mk/buildset.var

Add "myos" to OS_RANGE

Change OS from none to "myos"

Add the line:

```
CCFLAGS_PORT_OS_myos += -D__TMS_USE_STDDEF_FOR_SIZE_T=1
```

3.9 Building

Start a TTK shell using Windows START->MCCI->MCCI USB DataPump->Start TTK shell

If necessary, run your compiler setup script

Set *MYOS*_HOME:

```
export MYOS_HOME=c:\somewhere
```

Create a build directory:

```
cd \mcci\datapump\usbkern
makebuildtree –e –c armsdt portname
```

Then change directory to build/arm7/*portname*/*myos*-armsdt-checked

Then say:

```
bsdmake osmyos.a
```

This will build the OS library.

A full build might still not work; but the first step is to get the OS library working.

3.10 Problem Solving

3.10.1 OS Compilation Issues

If the target operating system has a complex, monolithic, target-specific configuration system, it may be impractical to compile the OS code in the MCCI environment. You then have two choices:

1. Compile the OS code in the target OS environment

2. Access all OS resources via the UMYOS_CONFIGURATION_INFO

In this case, the easiest way to proceed is to write all the code using the OS APIs. Then manually convert API calls into indirect function calls, convert #define'd constant references to references to constants in the UMYOS_CONFIGURATION_INFO.

Either convert structure references into method calls (i.e., call a function to find the value), or pass the absolute offset of the fields in the UMYOS_CONFIGURATION_INFO.

If there are structures that need to be referenced from the OS header files and from the MCCI header files, either duplicate the definition, or create an extra header file that can be included on either side.

4. Questions about MCCI Code

4.1 What does "BSS" keyword mean and why does MCCI use it?

BSS stands for "Block Started by Symbol". It comes from Unix. We use this keyword to mark variables that are being defined in a given module, but not initialized to any particular value.

The reason we use this is because of portability issues. Some embedded system C compilers need special instructions in order to create a variable. By using the `BSS` keyword, we ensure that we can portably declare a variable. (The definition of `BSS` changes depending on the compiler in use.)

4.2 What is `__TMS_CLOAKED_NAMES_ONLY` and why does MCCI use it?

MCCI header files use many typedefs and variable names, for example, `VOID`, `UINT16`, and so forth. As long as we are working in the pure MCCI environment, there is no problem.

Platform vendor header files also use many typedefs and structure names. These names may conflict with MCCI's names. pSOS is the best example. MCCI uses `CHAR` to mean the C "`char`" type. pSOS uses `CHAR` as a numeric constant.

The problem is this: how do we compile code that bridges from MCCI's world to the platform world?

MCCI's solution is to define all types twice. The first time, we define the type with a prefix, "`__TMS_`". These names are very ugly to look at and to enter, but they are very unlikely to collide with a platform-vendor symbol name. These type names are called the "cloaked" types, because they are effectively hidden from the platform header file's view.

If `__TMS_CLOAKED_NAMES_ONLY` is 0, then the header files immediately define the "public" form of the type based on the "cloaked names".

However, if `__TMS_CLOAKED_NAMES_ONLY` is 1, then the header files *only* define the "cloaked form".

To keep things consistent, MCCI uses a set of macros defined in `libport/i/def/typemacs.h`. These macros are used to generate all the cloaked and (optionally) the public types.

When compiling code in the os layer, we normally set `CLOAKED_NAMES_ONLY` to 1 before including any MCCI header files. This ensures that we don't accidentally interfere with the platform's header files.

MCCI coding rules require that we be very careful about converting from MCCI types to platform types, even if it seems "obvious" that they are the same. We have found that being careful is more reliable and robust.