DataPump Host Training Intro

Agenda

- Quick overview of MCCI's USB product architecture and implementation
  - Introduction & general organization →
  - Coding styles (emphasis on comments and code documentation) →
  - USBDI APIs →
  - DataPump Object System →
  - UPLATFORM →
    - Event handling →
    - Debug logging system →
  - USBD Initialization →
  - Library Facilities →

INTRODUCTION & GENERAL OVERVIEW

DataPump: What Is It?

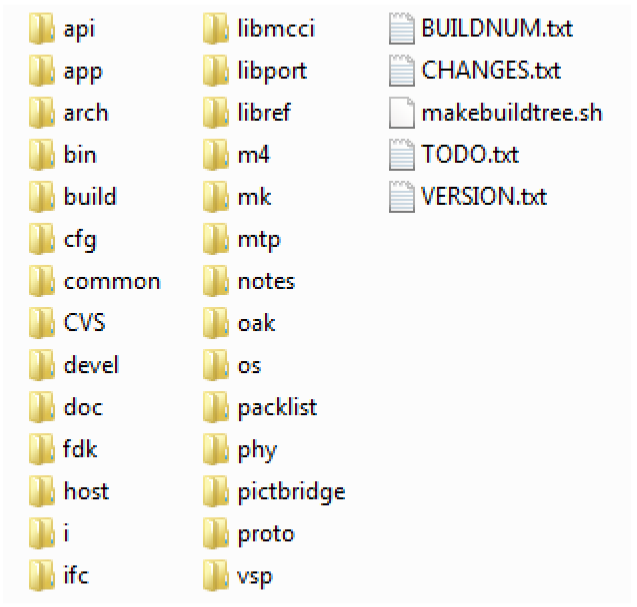
- MCCI's portable embedded USB framework
  - Add high-performance USB support to complex software environments
  - Supports creating USB host (Windows, embedded host and OTG)
  - Supports creating USB devices (low speed, full speed, high speed, super speed, multi-function)
  - Portable across
    - CPU, endianness, HW platform, OS, USB silicon, compiler, development platform
  - Zero-copy

DataPump Components

- Code
  - Device stack
    - Core (implements chapter 9 of the USB spec)
    - Device-class protocol modules (implement the relevant device class specs)
    - API modules
    - DCD (Device Controller Drivers) – low-level hardware interface code
  - Device-stack Applications (optional)
    - PictBridge, MTP
  - Host stack
    - HCD (Host Controller Drivers)
    - Host controller driver framework (HCDKIT)
    - USBD
    - Hub driver
    - Composite driver
    - Transaction Translator (TT) support
    - Class drivers (basic HID, Mass Storage)
    - Debug and verification code
- Tools and build system
  - USBRC – for devices, generates descriptors from higher-level device description
  - MCCI Build system tools
  - Debug and verification tools
  - Test applications

Code layout

- The top level directory is "usbkern", and it contains all the subcomponents (except the MCCI build tools)
- By default, code is compiled into directories under usbkern/build
- Each module has its own directory
- Library-style code, one function per file.



```
api       libmcci     BUILDNUM.txt
app       libport     CHANGES.txt
arch      libref      makebuildtree.sh
bin       m4          TODO.txt
build     mk          VERSION.txt
cfg       mtp
common    notes
CVS       oak
devel     os
doc       packlist
fdk       phy
host      pictbridge
i         proto
ifc       vsp
```

Quick glance at host stack

We switched to a view of a typical module

Making a build tree

- When you build the DataPump, all work happens “outside” the source tree, in a “build” tree
- Each build tree has a “pre-set” collection of settings: CPU, compiler, OS, target board, compile-time options
  - “target board” == “BSP” == [MCCI] “port”
- To set up a build tree, we use the “makebuildtree” command. This lives in datapump/usbkern (usbkern for short).
- By default, build trees will live under usbkern/build, but they can be anywhere.
- Simplest form is: makebuildtree {portname}, where portname is a known port
  - makebuildtree –L will list all the valid ports
- makebuildtree 1650
  - this sets up a checked build: i386/catena1650/w32-microsoft-checked

Source Tree

- All the sources are at datapump\usbkern
  - api – contains API wrappers for high-level views of firmware download, serial ports, mass storage
  - app – contains sample "application" code
  - arch – contains arch or target specific code
  - <u>common</u> – contains the core DataPump code for the device stack (and also contains some common code shared with the host stack)
  - doc – contains some text files that provide additional documentation
  - host – contains code that is specific for the embedded host stack
  - <u>i</u> – contains header files (.h files) – i is short for "include".

Source Tree (2)

- ifc – contains hardware-specific code. ifc stands for “interface”. DCDs are in the ifc directory; HCDs are located here, too. Some “PHYs” are here too.
- libmcci – contains non-USB-specific code (btree package, btree support, some general library functions such as CRC calculation)
- libport – library of header files for writing portable code – types and macros for defining types and type cloaking

Source Tree (3)

- libref – header files for “reference”
- m4 – macros for writing MCCI “portable assembly language”
- makebuildtree.sh – shell script for setting up build trees
- mk – build system makefiles
- mtp – sources for the MCCI MTP application
- os – root of the tree of os-dependent files
- phy – root of the tree of low-level code for controlling USB phys.

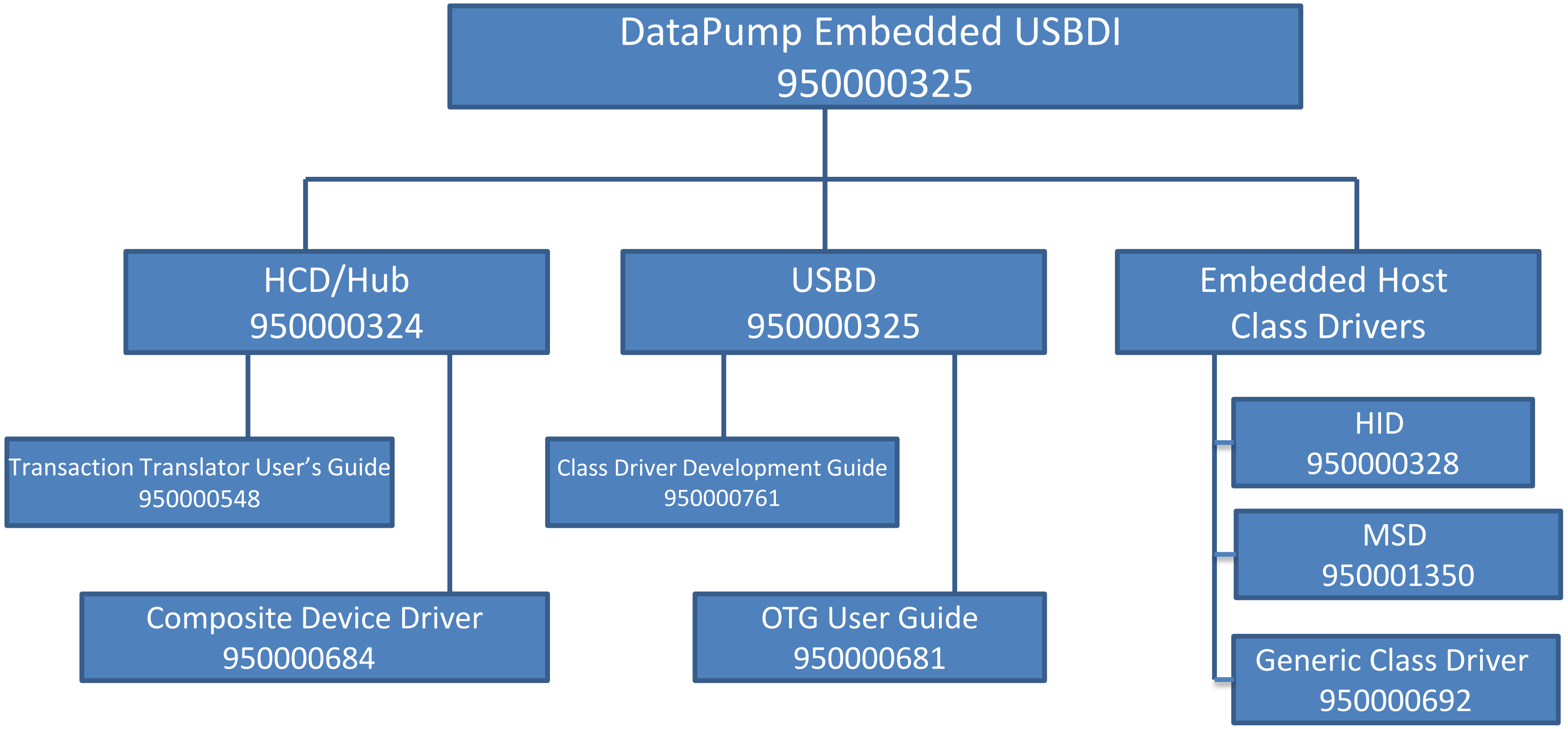
Source Tree (4)

- pictbridge – root of the tree of sources for MCCI's PictBridge product
- proto – DataPump "device-side" device class implementation code – AKA "protocol"
- VERSION.txt – contains the current version
- CHANGES.txt – repeated across the tree on a "module by module" basis

Version Numbers

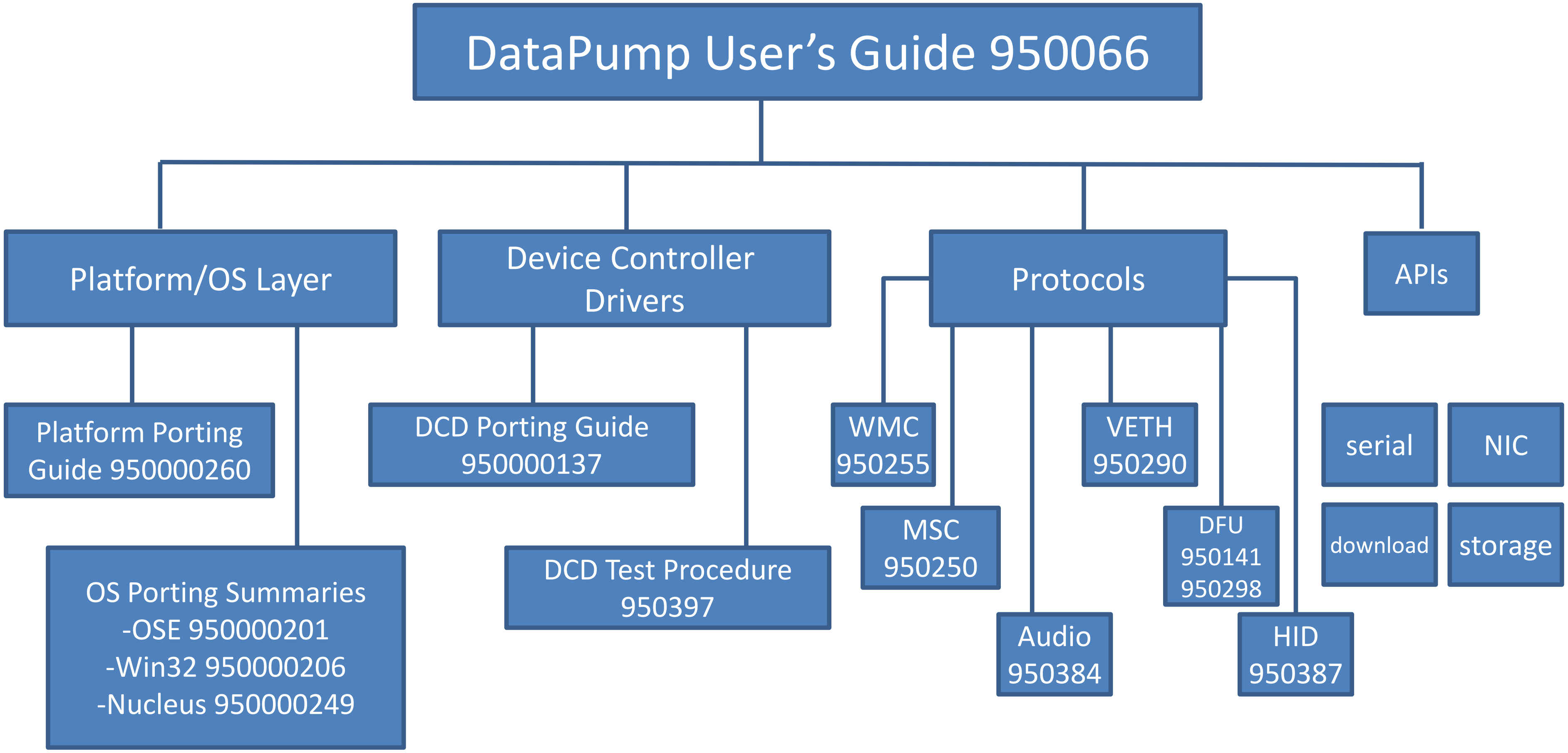
- Convention is that "production releases" should be "Vx.yy", and yy should end in an even digit. Internal releases are "Vx.yyz", and yy ends in an odd digit.
  - V3.22 is a released version of something
  - V3.25c is an internal build
  - V3.25 should never be created
  - V2.22c should never be created
- Patches to production releases are dotted ("V1.84.1", for example)
- Sequence is V3.24 < V3.25c < V3.26 etc.

Documentation Tree (1)



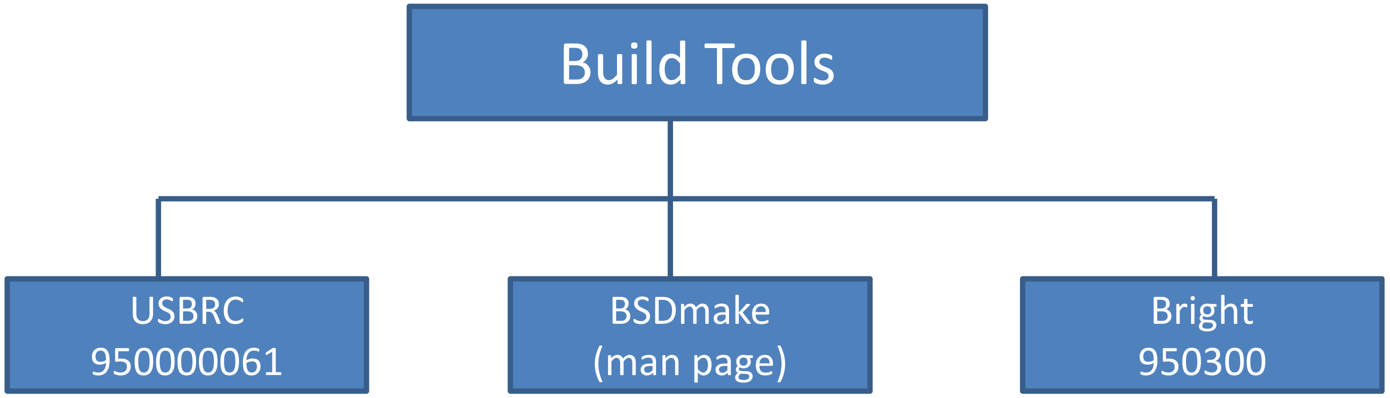
Documentation Tree (2)

MCCI USB DataPump Device Stack [for OTG and DRD] (the porting guides are useful for host, too)


DataPump User's Guide 950066
Platform/OS Layer
Device Controller Drivers
Protocols
APIs
Platform Porting Guide 950000260
OS Porting Summaries -OSE 950000201 -Win32 950000206 -Nucleus 950000249
DCD Porting Guide 950000137
DCD Test Procedure 950397
WMC 950255
MSC 950250
Audio 950384
VETH 950290
DFU 950141 950298
HID 950387
serial
NIC
download
storage

Documentation Tree (3)

Build Tools



Documentation Tree (5)

- Host verification test suite (HCDVT, USBDVT) -- optional
- Test tools (eg. WMCDVT, USBIOEX)
- Automatically extracted documentation (.chm files)

How to read (and write)
MCCI-style code

MCCI CODING STYLE

Introduction

- People encountering MCCI code (especially the header files) for the first time frequently experience considerable shock.
  - Why all the *!$? Directories?
  - Why all the *!$? Macros?
  - Why all the ugly prefixes on the types?

Why all the *!$? Directories?

- Or,

  “I can’t find anything!”
- The directory structure is based on a regular pattern, which is intended to express the relationships between the various files.
- The directory structure is intended to support a coding style that avoids conditional compiles

The Directory Structure

- Organizing Principles
- Important Places

Organizing Principles

- Alternation between “collection point” directories (“collections”) and “instance” directories (“places”).
  - if a dir contains source code, it shouldn’t contain directories
- *Important Levels of organization:*
  - *CPU Architecture*
  - *Platform implementation*
  - *Interfaces (USB device controller and host controller architectures)*
- *Parallel structure at each level*

Collections contain Places

- Normally the directories in a collection directory are specific “places”.
- Examples:
  - arch/arm7 contains the arm-specific code
  - os/none contains the os/none-specific code

Places can contain…

- Places can be terminal:
  - common/
  - i/
- Places can contain other places:
  - os/none/ contains os/none/common/
  - ifc/uss820/ contains ifc/uss820/common/
- Places can contain collections
  - arch/arm7/ contains arch/arm7/port/
- Places can contain files
  - but we try to restrict these to control info for places containing other places or collection points.

Some Directories are Hybrid

- In a few cases, directories can serve as both “places” and “collection points”.
- Most obvious example: libport’s compile control directories
  - libport/arch/arm7/mk/gcc contains the information for compiling for arm7 with gcc.
  - libport/arch/arm7/mk/diab contains the same information for using Diab.
- We try to avoid this, but sometime notational convenience outweighs formal considerations

Organizing Principles (2)

- *Alternation between "collection" directories and "instance" directories.*
- Levels of organization:
  - CPU Architecture
  - Platform implementation
  - Interfaces (USB device silicon architectures)
- *Parallel structure at each level*

CPU Architecture

- Primary distinction.
- Called “arch”
- Any “collection point” directory named “arch/” is used to organize code based on which instruction-set architecture is in use.
  - Same arch is used for different compilers
  - For multi-endian CPU families (arm, powerpc, etc.), we use the same code base and compile separate libraries (so arch’es don’t categorize based on endianness)
  - Sometimes arch-name is misleading (arm7, i386, ….)

Platform Implementation

- A specific combination of:
  - CPU
  - Compiler
  - Target board
    - USB device silicon
    - other chips
  - Operating System
  - …
- Called a “port”
- Equivalent to a “bsp”

Platform Implementation (2)

- MCCI-supplied “ports” are collected for each architecture
- Example: arch/arm7/port/wombat, arch/i386/port/catena1610, etc.
- Important – there can be multiple “applications” per port! Each one represents a different use of the DataPump on that hardware
- The setup information from the port comes from variables defined in the file arch/…/port/…/mk/buildset.var
- IMPORTANT: no need for port/ directory to live under usbkern.

Organizing Principles (3)

- *Alternation between “collection” directories and “instance” directories.*
- *Levels of organization:*
  - *CPU Architecture*
  - *Platform implementation*
  - *Interfaces (USB device silicon architectures)*
- Parallel structure at each level

Parallel Structure at Each Level

- Example: the “mk” directory contains info that controls builds. Information can be divided into usbkern global, libport global, etc….

  usbkern/mk

  usbkern/libport/mk

  usbkern/arch/arm7/mk

  usbkern/os/none/mk

  usbkern/ifc/uss820/mk

  usbkern/libport/arch/arm7/mk/

  usbkern/arch/arm7/port/wombat/mk

Important Places

- */mk: holds compilation information
- */i: holds include files (for C)
- */def: holds assembly-language header files. (note that libport, for historical reasons, is */i/def).
- */common: contains the source at a given level (not */src, because everything is source!)
  – usbkern/common has the top-level sources

Important Collections

- */arch: holds a collection of CPU-specific architectures.
- */os: holds a collection of os-specific directories
- */proto: holds a collection of protocol-specific modules
- */ifc: holds a collection of USB-hardware-interface-silicon-specific directories (host and device hardware)
- */port: holds a collection of platform-specific implementations

Interfaces: USB Silicon

- Code for handling specific USB silicon code is collected at the “ifc” collection point.
- DataPump builds a library for each different kind of silicon
- There may be some compilation options having to do with how the registers are accessed
- Such options are encapsulated by macros and controlled by “hilbusio.h” (a virtual include file)

Why all the *!$? macros?

- “Because we care”
- The intent is to allow you to upgrade without modifying existing code
- This allows us to improve our code without breaking yours
- We also want you to be able to integrate our code without editing it
  - again, this allows for updates

Major Kinds of Macros

- Type cloaking
- Structure Initialization
- Boilerplate for protocols

Type Cloaking

- Problem: everybody likes short names; nobody can agree on them.
  - Example: pSOS defines CHAR as an enumerated constant.
- Type cloaking allows most MCCI code to be written using sane-looking types (CHAR, VOID*, etc.)
- So most code looks very readable
- However code that needs to be mixed with customer or OS headers has to use the longer (ugly) names
  - Long names are of the form __TMS_{whatever} and are always in scope
  - Short names can be hidden selectively on a file-by-file basis
  - Header files look horrible!

Type Cloaking Basics

- Rather than using typedef directly, use special macros:
  - **`__TMS_TYPE_DEF(`***type_name***`,`** *type_expr***`);`**
    defines a simple type *type_name* in terms of *type_expr*.
  - **`__TMS_TYPE_DEF_STRUCT(`***struct_type_name***`);`** defines a structure type. The body is defined separately .
  - **`__TMS_TYPE_DEF_UNION(`***union_type_name***`);`**
    defines a union type. The body is defined separately.

Created Type Names

- Each such macro defines two types, *type_name* and P*type_name.*
- `__TMS_TYPE_DEF()` is roughly equivalent to:

  typedef *type_expr* *type_name*, ***P***type_name*;**

- Example:

  `__TMS_TYPE_DEF(FOO, char)` is roughly equivalent to:

  typedef char FOO, *PFOO;

- In addition, auxiliary *cloaked* names are created, by adding a prefix (**`__TMS_`**) to each of the types, effectively:

  typedef *type_expr* **__TMS_***type_name***,**
  __TMS_Ptype_name***;**

- Example:

  `__TMS_TYPE_DEF(FOO, char)` also expandes to:

  typedef char __TMS_FOO, *__TMS_PFOO;

Created Type Names (2)

- Example:
  - __TMS_TYPE_DEF(UINT8, unsigned char);
  - Main typedef names:
    - typedef unsigned char UINT8, *PUINT8;
  - Auxiliary (cloaked) typedef names:
    - typedef unsigned char __TMS_UINT8, *__TMS_PUINT8;

How Cloaking Works

- When cloaking is enabled for a given compilation, *only* the types with the **`__TMS_`** prefix are defined. Plain types are not defined.
  - so in our first example, __TMS_FOO and __TMS_PFOO are defined, but FOO and PFOO are <u>not</u> defined
- Therefore the plain types, which might clash with another software component's usages, are hidden when cloaking is enabled
- However, they can still be used in their (uglier) prefixed form, both to define other types, and to write code that has to include header files from two components.

Watch Out For…

- The *type_expr* in a definition must be given in terms of visible types.
  - If file is never used with cloaking on, these don’t need to be prefixed
  - But if file is to be used with cloaking turned on, you must prefix *all* MCCI types and types defined with the __TMS_TYPE_DEF() family of macros
    - exception: __TMS_TYPE_DEF_ARG() is special and deals with “widening”, see below
- Enum values and #define names also are in the name space and may clash with other uses.
  - Cloaking for these has to be done manually by repeating the definitions twice: first the ugly form, then (#if !__TMS_CLOAKED_NAMES_ONLY) the simple form
  - We have a script, uncloak_defs, that makes it a lot easier for #defines.

Defining Function Types

- Provides cloaking and standard pointer references

  ```
  __TMS_TYPE_DEF_FUNCTION(
              type_name,
              function_return_type,
                          ( arguments )
              );
  ```

- Equivalent to:

  typedef *function_return_type* (*type_name*)(*arguments*);
  typedef *type_name* ***P***type_name***;**

- If *type_name* is **`MY_FN`**, then:
  - The types **`MY_FN`** and **`PMY_FN`** are created.
  - "**`MY_FN foo;`**" declares "**`foo`**" to be a function with the specified prototype – **`foo`** is a function, not a pointer to a function.
  - "**`PMY_FN pMyFn;`**" declares "**`pMyFn`**" to be a pointer to a function (such as "**`foo`**") with the specified prototype.

- The macro **`__TMS_FNTYPE_DEF()`** is a deprecated synonym for **`__TMS_TYPE_DEF_FUNCTION()`**

Function Types and Cloaking

- As usual, types "**`__TMS_`***type_name*" and "**`__TMS_P`***type_name*" are created.
- If cloaking is turned on, *only* the **`__TMS_`**… types are defined
- If cloaking is to be used, all arguments in the *argument_list* must be declared using cloaked types.
  - Otherwise header file won't compile when cloaking is turned on.

Function Types and Cloaking (example)

- Here's a sample declaration:

```
__TMS_TYPE_DEF_FUNCTION(
  MYFNTYPE,
    __TMS_BOOL,
        (
        int /* x */,
        __TMS_UINT32 /* y */
        ));
```

- Then you can do the following:
  - in the header file, to get a prototype:

```
__TMS_MYFNTYPE MyFunction;
```

  - then in the implementation file:

```
BOOL MyFunction(
    int arg1,
    UINT32 arg2
    )
    { /* body */ }
```

Why Use Function Types

- A method is declared in an include file, and your function is an instance of a method.
- Problem: the abstract type of the method is changed
  - With explicit prototypes, you have to change all .h files defining prototypes for functions that implement that method, as well as all the .c files containing the functions themselves
  - With function types, only the .c files need to be changed. The correspondence is immediately shown in the file that has to be changed when you recompile

C and “widening” (Argument Promotion)

- The C standard says that each type occurs in two forms: the normal form and the widened form.
- The normal form is what’s used in most cases (functions with prototypes, variables, structure entries, etc.)
- However, if you have a function that takes a variable number of arguments, the type of a variable changes automatically when passed to the function
  - This is called “argument promotion” in the C standard

C and “widening” (2)

- For example, suppose you have:

  -
    ```
    char c;
    printf("%c", c);
    ```
- Printf’s prototype is **`int printf(const char *fmt, …);`**
- So ‘c’ is converted to an int (or unsigned int) prior to passing to printf() – compiler can decide
- Later in printf(), we must write **`va_next(ap, type)`**. But this *type* needs to be the *promoted* type, not the *normal* type; and we don’t know if it’s unsigned or signed.
- So how do we code printf() portably?
  - Bear in mind, our rules require us to avoid use of #if in C code wherever possible.

C and “widening” (3)

- Libport has a solution.
- For each type, libport can define an auxiliary type, **`ARG_`*`type`***, which represents the type used by the C compiler when promoting *type.*
- printf() can then say **`va_next(ap, ARG_CHAR)`**, and the right code will be emitted

ARG_ prefixes and widening

- Problem: C silently widens some types when passing to varargs functions
- We may know the original type; what should we use for *type* in **va_arg(ap, *type*)**?
  - For UCHAR, USHORT, should use unsigned; for SCHAR, SHORT, should use int
  - But what about abstract types? We will inadvertently embed knowledge about the concrete representation in our code!
- Solution: when creating a type, also create an ARG_*type*, that is the result of *widening* type according to the standard rules.

ARG_ prefixes and the type macros

- For all the standard types created by libport, ARG_*type* and PARG_*type* variants are created.
  - Necessary to preserve abstraction.
- For application-specific types that actually (may) change under widening, use:
  __TMS_TYPE_DEF_ARG(*type_name, base_type_name*);
  - Notice that you must use one of the predefined types that has an ARG_ prefix; you can't use "unsigned char", but must use "UCHAR" or "UINT8".
  - Also, the "__TMS_" prefix is *never* used in the base_type_name or the type_name; it's automatically supplied.

Usage of ARG_

- To define EXTENDC as UINT16, with proper definition of ARG_EXTENDC:

```
__TMS_TYPE_DEF_ARG(
          EXTENDC, UINT16
          );
```

- This creates EXTENDC, PEXTENDC, ARG_EXTENDC, PARG_EXTENDC, and __TMS_ equivalents
- Since structures, unions, and functions are not interchangeable with scalar types, no ARG_ prefix is defined for these types. So to this degree your code reflects design decisions that a given type is or isn't scalar.

Manual Cloaking

- For Enums and #defined symbols and macros, manual cloaking must be used if necessary
- Simple rules:
  - Unconditionally define the item using the "__TMS_…" name first

    ```
    #define __TMS_MY_MACRO(x,y) something
    ```

  - Then, define the uncloaked name in terms of the cloaked name, inside #if bracketing:

    ```
    #if !__TMS_CLOAKED_NAMES_ONLY
    # define MY_MACRO(x,y)    \
                 __TMS_MY_MACRO(x,y)
    #endif /* !__TMS_CLOAKED_NAMES_ONLY */
    ```

  - MCCI has a simple awk/shell script that makes this easy, if your editor can filter regions. (uncloak-defs.sh)

Structure Initialization

- Problem: C-89 structure initialization is structural. (C99 is better, but…)
  - if element order changes, every initializer must change
  - if elements are added, new initialization must be added
  - if elements are inserted or reordered, every initialization must be checked – might compile but not work

Structure Initialization Solution

- MCCI defines macros to do initialization for you
- The macro NAMES have version numbers.
- So if you code a macro with a “V1” name, and we later change the macro, we will call it “V2”, and change the “V1” definition so that it does the right thing
- You don’t have to upgrade your code at all, because we’ll make sure that the old V1 macros call the V2 macros with the right arguments

Structure Initialization Macros

- Macro maps arguments onto required initialization sequence, including leading, trailing ‘{‘ and ‘}’.
- Macro name contains explicit version (_V1, _V2, …)
- MCCI will make best efforts to ensure that code written with _V1 macros can be compiled in future.
  – To take advantage of _V2, _V3, … features, you have to edit your code. But old code still compiles and runs

Structure Initialization Macros – Rules

- If you are making your own, initially define your macro as …_INIT_V1().
- After initial development, consider the _V1() macro *header* frozen. You can, of course, change how it expands.
- If new elements are added, add a new version of the macro (_V2, _V3, etc.); then change the _V1 macro so that it's written in terms of the _V2 macro.

Runtime Structure Initialization

- Direct initialization of structures doesn’t suffer the positional problem – you can write:

  ```
  pFoo->x = v1; pFoo->y = v2;
  ```

  and for sure, x and y are set, no matter how FOO changes
- Problem is if we add new fields to FOO.
- So we have a similar family of runtime init macros, called {structname}_SETUP_V1(), _V2(), etc.
- First arg is pointer to object, remaining args are the same as _INIT_Vx.
- What these do is essentially change from “direct access” to “object oriented setup method”

Libport

- The key to MCCI-style portability
- “Quasi object-oriented include files”
- All binding between abstract code and compiler and processor concepts happens in libport.
- Libport headers are always included as “def/*name*.h”
- Header file names are abstract
- Originally developed for TrueTask, an MCCI operating system project
- Normally you don’t have to touch anything here

Libport Structure

- Similar to DataPump
  - i/def/* -- portable .h files, used on any target (the “def/” is historical accident)
  - mk/* -- files used by MCCI build system
  - arch/{arch}/* -- files specific to a given target architecture
    - arch/{arch}/i/def/* -- .h files specific to the selected architecture
    - arch/{arch}/mk/{compiler}/* -- BSDmake files specific to the selected architecture/compiler combination
    - arch/{arch}/mk/{compiler}/sys.mk – the starting point for all BSDmakes for this target (configured by makebuildtree)

USBDI APIS

USBD Architecture Overview

- Foundation: chapter 10 of the USB 2.0 Specification
  - *At this point, we'll break from these slides to walk through the contents of section 10.1*

USB standard software mechanisms

- 10.5.1.1 USBD Initialization
  - MCCI's USBD is initialized at system start up time; we'll describe this in the initialization & setup section
- 10.5.1.2 USBD Pipe Usage
  - Pipes are modeled to client drivers by USBPUMP_USBDI_PIPE_HANDLE objects.
    - Pointer-sized objects
    - These are created when a configuration or interface is selected.
    - These are disposed on removal, or when the configuration or interface is changed.
    - The underlying data structure inside the USBD is not visible to clients
  - A special pipe handle is reserved for the default pipe. This handle is valid as long as the device is connected to the system

Communicating with Devices

- All devices are connected to the system on unique “ports”.
- To communicate with a given device instance, clients send messages to the port associated with that device
- Ports are data objects created by USBD, and partially shared with clients. Name USBPUMP_USBDI_PORT, “host/i/usbpump_usbdi_port.h”.
- When a device arrives, a class driver instance is located and bound to the port; and the class driver new-instance notification entry point is called, passing the a pointer to the port, a pointer to the allocated or recycled class driver instance, and a “port key” which represents the connection between class driver and that instance of the device.

Port Keys and Hot Plugging Devices

- Ports conceptually have longer lifetimes than devices. If a device plugged into a port is removed, and a new device is plugged in, the new device will use the same port as the old device.
- To prevent requests for the old device from being passed to the new device, each time a device is found, a 32-bit **port key** is created and assigned to that session of the device. The class driver is given that key when it is bound to the port. All requests from the class driver to the port are accompanied by the port key.
- When an device is unplugged, the port key is changed to a new value. Any request that arrives with the old value (or with a value that doesn’t match the current key) is immediately completed with error status.

Class Driver Flow (initialization)

- At initialization time, the class driver object is created
- The class driver object has associated device matching info strings, to match to devices by class, by vid/pid, etc.
- The class driver pre-allocates one or more function objects (USBPUMP_USBDI_FUNCTION); on embedded systems, there's one of these for each device that can be simultaneously supported
  - If you have only one function defined for mass storage, then even if you have a hub and several thumb drives plugged in, only one will operate
  - The number is defined in the class-driver's initialization structure USBPUMP_USBDI_DRIVER_CLASS_CONFIG::NumInstances.
  - (see host/i/usbpump_usbdi_init.h)
- When the class driver creates each USBPUMP_USBDI_FUNCTION, it associates an object-IOCTL function with the instance of the function – so that function object can receive messages. We use that capability to announce arrival and departure..

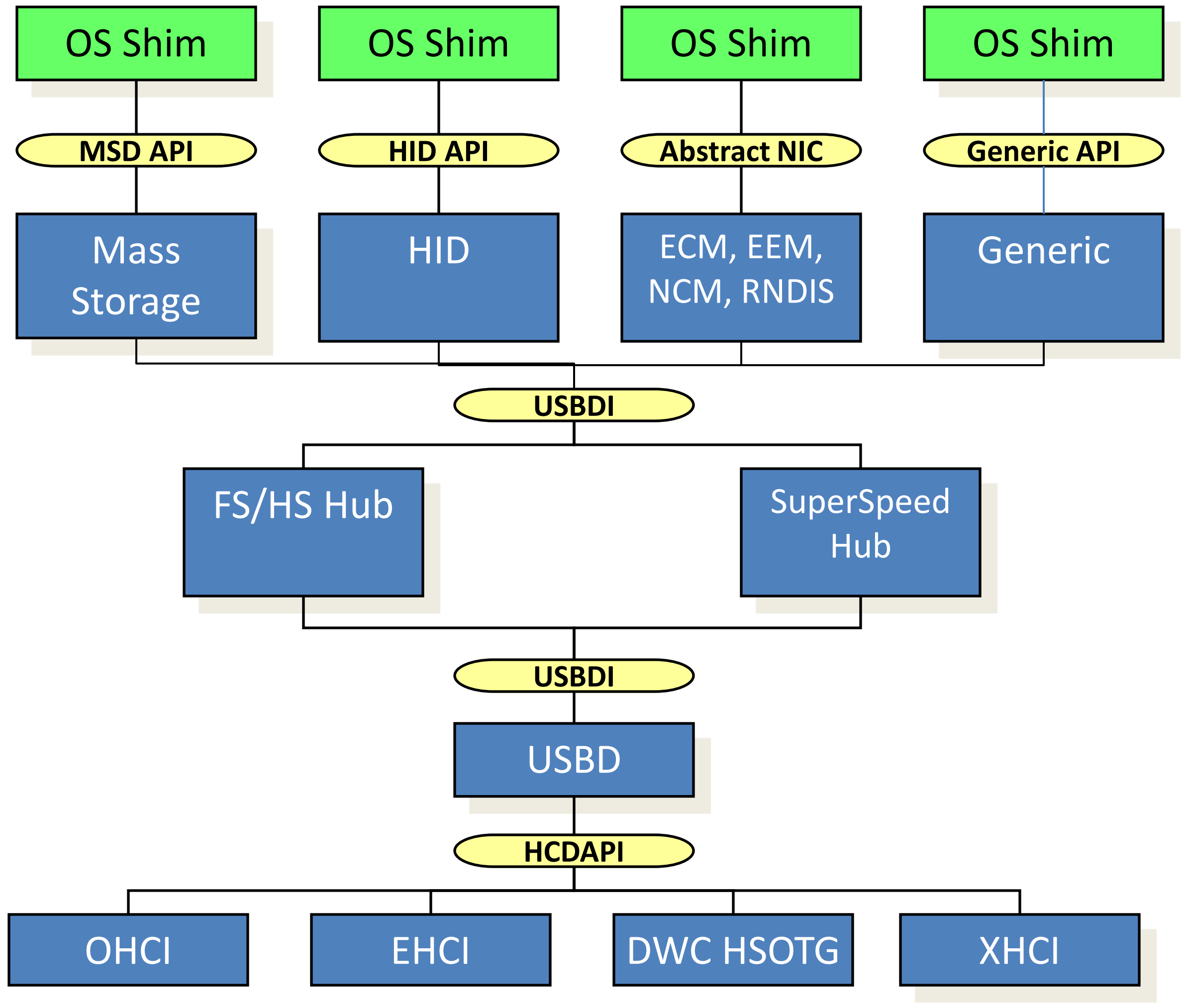
Class Driver Flow (arrival)

- When a new device arrives and has been enumerated, USBD scans all the known class drivers looking for a match. Priorities are used for resolving matches (vid/pid is higher priority than just a class match, for example)
- Once a class driver is selected, a USBPUMP_USBDI_FUNCTION is pulled from the pool associated with that driver, and associated with the USBPUMP_USBDI_PORT to which the device is connected.
  - pFunction->Function.PortInfo is initialized with all the info needed to talk to the instance. In complex systems with multiple host controllers, this info might be different depending on where the device is connected. In simple systems, this is always the same.
    - The class driver framework also associates an abstract pool with the function. This allows for memory to be pre-assigned to the function, and prevents leaks
  - An IOCTL (USBPUMP_IOCTL_EDGE_USBDI_INSTANCE_ARRIVAL) is sent to the function object.
  - The function object code starts the event-driven sequence of querying descriptors, setting configuration, starting operation, and (if appropriate) notifying clients.

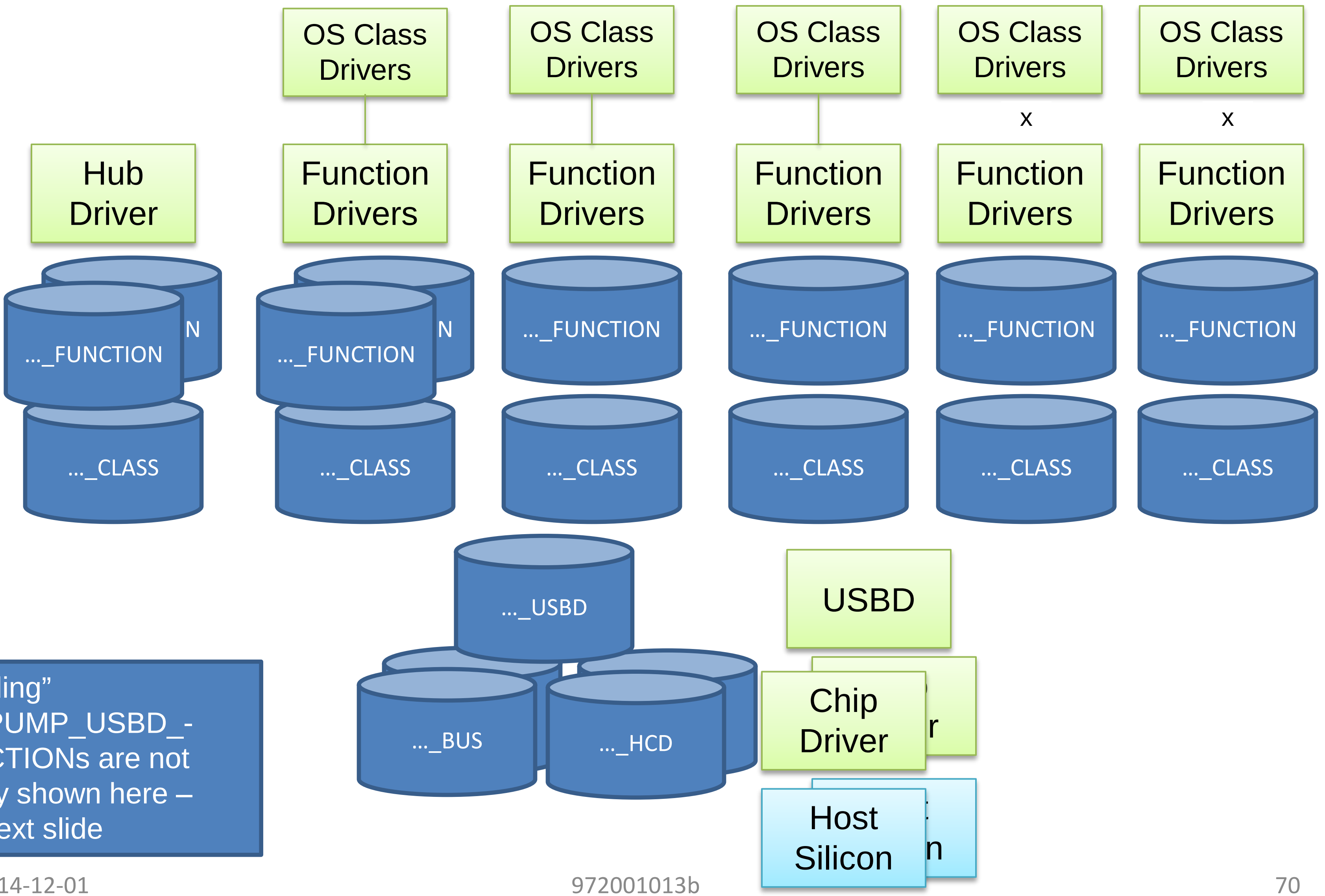
Class Driver Flow (departure)

- On hot unplug, something similar happens.
  - USBD changes the port key to a new value, blocking any further I/O to the device
  - USBD sends USBPUMP_IOCTL_EDGE_USBDI_INSTANCE_DEPARTURE_ASYNC to the function object.
  - This is an asynchronous IOCTL; so the function driver can cancel URBs, shut down any complex finite state machines. When everything is idle the function driver completes the IOCTL.
  - USBD then unlinks the function object from the port, and moves the USBPUMP_USBDI_FUNCTION back to the pool of free function objects associated with the class driver.
  - All memory allocated from the function’s abstract pool is automatically released at this moment (if it hasn’t already been freed)

Driver Architecture

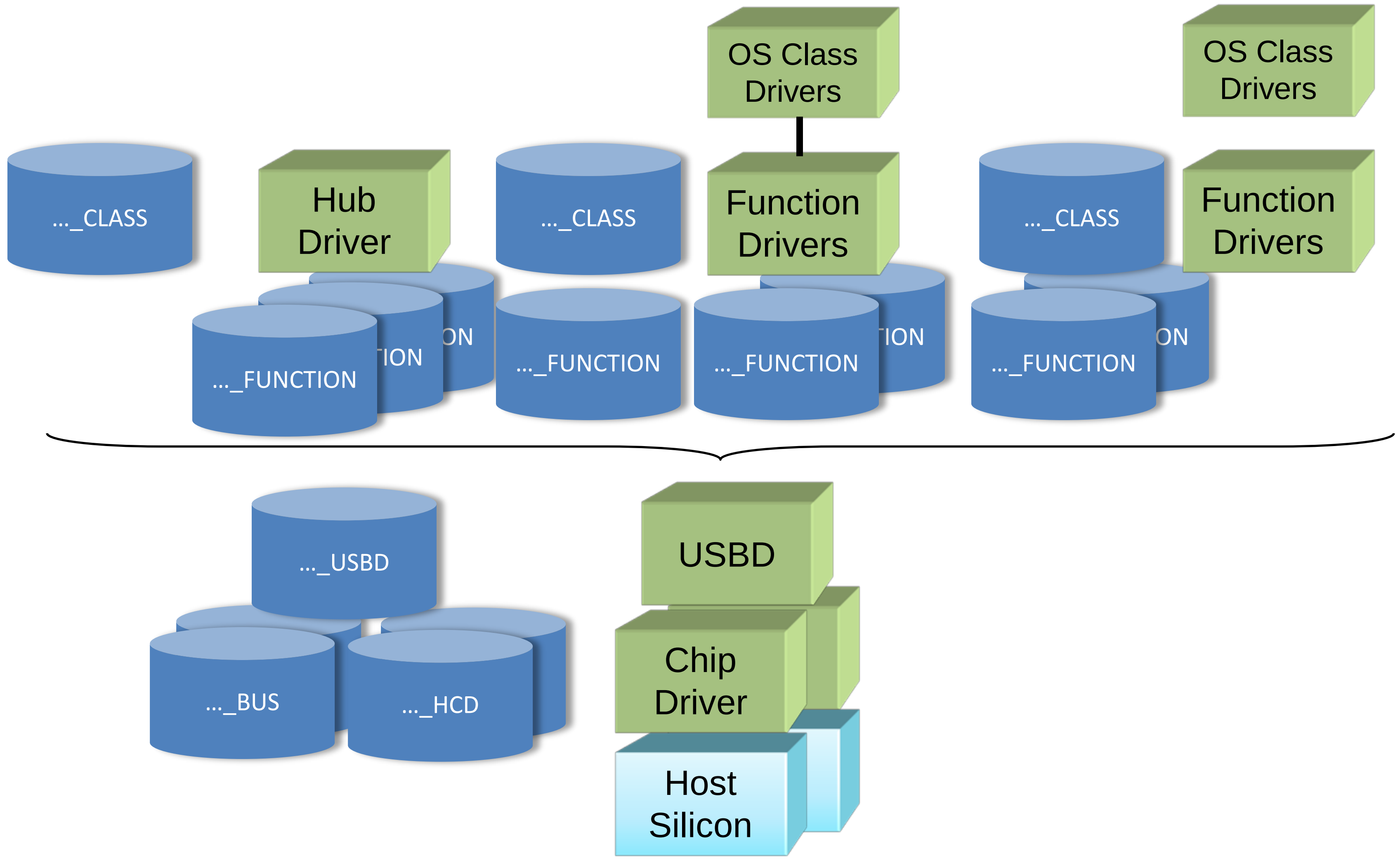


Code/Object Hierarchy



“Pending” USBPUMP_USBD_-FUNCTIONs are not clearly shown here – see next slide

Function Parking Diagram



Communicating with USBD

- Clients typically communicate with devices under USBD using “USB Request Blocks”, or URBs. These are blocks of memory allocated by the client, and passed by reference to USBD
  - named USBPUMP_URB
  - Defined in host/i/usbpump_usbdi_request.h
- All requests are asynchronous, meaning that the request is submitted, and later a completion callback function is called to signal that the request has been processed.
- All requests **must be made from within DataPump context**. All callbacks will be made in DataPump context

Sending URBs

- Define a completion function:

```
USPBUMP_URB_DONE_FN MyCompletionFunction;

VOID MyCompletionFunction(
             USBPUMP_USBDI_PORT *pPort,
             USBPUMP_URB *pUrb,
             VOID *pDoneInfo,
             ARG_USTAT Status
             )
```

- To submit the URB:

```
// you have these two things from attach time
USBPUMP_USBDI_FUNCTION *pFunction;    // device is attached to this port
USBPUMP_USBDI_PORT_KEY PortKey = pFunction->PortInfo.PortKey;
                                                  // this is the “generation number”
USBUMP_USBDI_PORT *pPort = pFunction->PortInfo.pPort;

// allocate the URB
sizeUrb = pFunction->Function.PortInfo.UrbTotalSize;
pUrb = UsbPumpPool_Alloc(pFunction->Function.pPool, sizeUrb);

// build the URB, for example:
UsbPumpUrb_PrepareGetDeviceInfo(pUrb, sizeUrb, PortKey, /* timeout */0);

// submit the URB
(*pPort->Port.pMethods->pSubmitRequest)(
             pPort, pUrb, MyCompletionFunction, /* context */ pDoneInfo
             );

// now, don’t touch the URB until the completion callback happens.
```

Standard Commands

- MCCI's USBD doesn't have lots of URBs for standard commands. Instead, it monitors control transfers to the default pipe of the device, and intercepts any standard (chapter 9) commands.
- Any work needed in USBD to correctly implement the command is performed automatically
- For IN and OUT commands with non-zero wLength, **`pUrb->UrbControlIn.nBuffer`** must be set to be at least as large as **`pUrb->UrbControlIn.Setup.uc_wLength[0:1]`**

10.5.2.1 Interface State Control

- All interface state changes are made by sending a USBPUMP_URB to the device via the USBPUMP_USBDI_PORT. The URB specifies the pipe handle and the request code.
- The available URBs are:
  - USBPUMP_URB_RQ_ABORT_PIPE
  - USBPUMP_URB_RQ_RESET_PIPE
- With RESET_PIPE, you can optionally send a clear-feature to the device, or clear the host data toggle, as well as clearing the endpoint halted state.

10.5.2.3 Getting Descriptors

- On the bus, getting descriptors is a CONTROL operation targeting endpoint 0 of the device
- With MCCI’s USBDI, you create a USBPUMP_URB_RQ_CONTROL_IN URB, and fill in the setup packet. USBDI notices that this is a standard command, and does whatever extra operations are needed for getting the requested descriptor.
  - In particular, the composite driver will filter the CONFIG bundle so that each child function driver only receives descriptors relevant to that driver

Getting Current Configuration Settings

- In an embedded system, we don’t want to spend the memory to support dynamic queries – the class driver must remember what settings were chosen most recently.
- When choosing a setting, the class driver normally uses the SUGGEST_CONFIG URB, which causes the USBDI to parse a supplied configuration descriptor (or set of descriptors) and build a tree of configurations, interfaces, endpoints, pipe handles, etc.
  - SUGGEST_CONFIG’s parser is very carefully coded to be robust against non-compliant devices. It’s much better to use SUGGEST_CONFIG than to try to parse the descriptor yourself
- After using SUGGEST_CONFIG to build a configuration tree, the class driver may modify the tree.
- Then the driver passes the tree to USBD using DEFINE_CONFIG
  - If the device has alternate settings, you may optionally reserve bandwidth for the worst-case setting when you do the DEFINE_CONFIG. This has the advantage of guaranteeing the SET_CONFIG and SET_INTERFACE will work.
  - Otherwise, bandwidth is not reserved until you actually select the configuration and any alternate settings using SEND_CONTROL_OUT_URB(SET_CONFIG) or …(SET_INTERFACE). But SET_CONFIG or SET_INTERFACE may fail due to bandwidth restrictions.

Managing Status

- USBPUMP_URB_GET_PORT_STATUS
  - Returns a bit mask
  - ENABLED means that the port is enabled by the hub driver; if not enabled, then the port is disabled (perhaps due to a babble error)
  - CONNECTED means that a device is connected to the port.
- USBPUMP_URB_RQ_CONTROL_IN(GET_STATUS) is used to fetch status of devices, interfaces or pipes. You simply build the standard GET_STATUS request indicating the component for which status is required

Sending Class and Vendor Commands

- Class commands are sent using USBPUMP_URB_RQ_CONTROL_{IN, OUT}.
- You fill in the SETUP packet with the info you want to send. For the data phase, you provide a pointer to the buffer you want to send to or fill from the device.

Establishing Alternate Settings

- Use USBPUMP_URB_RQ_CONTROL_OUT(SET_INTERFACE) with the interface number and the alternate setting you want. USBD will automatically deactivate the pipes on the old alternate setting, and activate the pipes on the new alternate setting
- Device must already be configured.

Establishing Configuration

- See “Getting Current Configuration Settings”

Setting Descriptors

- This is almost never done in practice
- However, if you need this, use USBPUMP_URB_RQ_CONTROL_OUT(SET_DESCRIPTOR) and provide the descriptor data in a buffer
- The USBD doesn’t use the descriptor data in any way; if you want to re-enumerate the device, the class driver must send the appropriate request, USBPUMP_URB_RQ_REENUMERATE_PORT

BULK and INTERRUPT pipe transfers

- These are handled the same way
- Use USBPUMP_URB_RQ_BULKINT_IN or USBPUMP_URB_RQ_BULKINT_OUT
- Provide a VOID * pointer to the buffer (for BULKINT_IN) or CONST VOID * pointer (for BULKINT_OUT).
- You optionally supply a **buffer handle** – this is arbitrary system-specific context for the buffer, provided by the UHILAUX. On systems with MMUs, this typically holds a system-specific scatter/gather list. On systems with restricted DMA, this may be used for tracking DMA bounce buffers.
  - Normally, if your code is initiating the transfer, the buffer handle is NULL, and you don’t need to worry about this.
  - If integrated into a bigger OS, with OS API emulation, the API emulation layer will provide the handle.

BULK and INTERRUPT pipe transfers (2)

- Preparing the URB:
  - VOID UsbPumpUrb_PrepareBulkIntIn(
    USBPUMP_URB *pUrb, BYTES UrbLength, USBDI_PORT_KEY PortKey, USBDI_TIMEOUT TimeOut,
    USBDI_PIPE_HANDLE hPipe,
    VOID *pBuffer, BYTES nBuffer, UINT32 TransferFlags
    );
  - VOID UsbPumpUrb_PrepareBulkIntOut(
    USBPUMP_URB *pUrb, BYTES UrbLength, USBDI_PORT_KEY PortKey, USBDI_TIMEOUT TimeOut,
    USBDI_PIPE_HANDLE hPipe,
    CONST VOID *pBuffer, BYTES nBuffer, UINT32 TransferFlags
    );
  - VOID UsbPumpUrb_PrepareBulkIntInV2(
    USBPUMP_URB *pUrb, BYTES UrbLength, USBDI_PORT_KEY PortKey, USBDI_TIMEOUT TimeOut,
    USBDI_PIPE_HANDLE hPipe,
    VOID *pBuffer, BYTES nBuffer, USBPUMP_BUFFER_HANDLE hBuffer, UINT32 TransferFlags
    );
  - VOID UsbPumpUrb_PrepareBulkIntOutV2(
    USBPUMP_URB *pUrb, BYTES UrbLength, USBDI_PORT_KEY PortKey, USBDI_TIMEOUT TimeOut,
    USBDI_PIPE_HANDLE hPipe,
    CONST VOID *pBuffer, BYTES nBuffer, USBPUMP_BUFFER_HANDLE hBuffer, UINT32 TransferFlags
    );

BULK and INTERRUPT transfer flags

- The following flags are defined for use by customer code:

USBPUMP_URB_TRANSFER_FLAG_ASAP

For ISOCH transfers only:

USBPUMP_URB_TRANSFER_FLAG_SHORT_OK

USBPUMP_URB_TRANSFER_FLAG_LINKED

USBPUMP_URB_TRANSFER_FLAG_POST_BREAK

USBPUMP_URB_TRANSFER_FLAG_FILTER_EXTRA_DESC

CONTROL transfer flags

- The following flags are defined for use by customer code:

USBPUMP_URB_TRANSFER_FLAG_SHORT_OK

For control-IN only – a short transfer is not considered an error.

USBPUMP_URB_TRANSFER_FLAG_FILTER_EXTRA_DESC

For SuperSpeed get-descriptor commands only: USBD will remove any SuperSpeed-specific descriptors. This is useful for backwards compatibility with legacy device drivers that don’t comprehend the extra descriptors.

CONTROL pipe transfers

- This includes the default pipe, and in fact is primarily used for the default pipe
- Use USBPUMP_URB_RQ_CONTROL_IN or USBPUMP_URB_RQ_CONTROL_OUT
- Provide a VOID * pointer to the buffer (for CONTROL_IN) or CONST VOID * pointer (for CONTROL_OUT).
- Provide an 8-byte setup packet, formatted in USB wire-level byte order (use USETUP_WIRE type)
- Use NULL as the pipe handle for the default pipe, or use the pipe handle returned by DEFINE_CONFIG for non-default pipes.
- Use UHIL_LE_PUTUINT16() for setting up wLength, wValue, wIndex.

Control pipe transfers (2)

- Preparing the URB:
  - VOID UsbPumpUrb_PrepareControlIn(
    USBPUMP_URB *pUrb, BYTES UrbLength, USBDI_PORT_KEY PortKey, USBDI_TIMEOUT TimeOut,
    USBDI_PIPE_HANDLE hPipe,
    VOID *pBuffer, BYTES nBuffer, UINT32 TransferFlags,
    CONST USETUP_WIRE *pSetup
    );
  - VOID UsbPumpUrb_PrepareControlOut(
    USBPUMP_URB *pUrb, BYTES UrbLength, USBDI_PORT_KEY PortKey, USBDI_TIMEOUT TimeOut,
    USBDI_PIPE_HANDLE hPipe,
    CONST VOID *pBuffer, BYTES nBuffer, UINT32 TransferFlags,
    CONST USETUP_WIRE *pSetup
    );
  - VOID UsbPumpUrb_PrepareControlInV2(
    USBPUMP_URB *pUrb, BYTES UrbLength, USBDI_PORT_KEY PortKey, USBDI_TIMEOUT TimeOut,
    USBDI_PIPE_HANDLE hPipe,
    VOID *pBuffer, BYTES nBuffer, USBPUMP_BUFFER_HANDLE hBuffer, UINT32 TransferFlags,
    CONST USETUP_WIRE *pSetup
    );
  - VOID UsbPumpUrb_PrepareControlOutV2(
    USBPUMP_URB *pUrb, BYTES UrbLength, USBDI_PORT_KEY PortKey, USBDI_TIMEOUT TimeOut,
    USBDI_PIPE_HANDLE hPipe,
    CONST VOID *pBuffer, BYTES nBuffer, USBPUMP_BUFFER_HANDLE hBuffer, UINT32 TransferFlags,
    CONST USETUP_WIRE *pSetup
    );

ISOCH transfer flags

- The following flags are defined for use by customer code:

USBPUMP_URB_TRANSFER_FLAG_ASAP

> If set, the ISOCH operation is scheduled at the next possible opportunity. If reset, the frame number from the URB is used as the start point for the URB.

USBPUMP_URB_TRANSFER_FLAG_SHORT_OK

> For IN pipes, a short transfer is not considered an error

DATAPUMP OBJECT SYSTEM

Common properties of objects

- All objects can be found (enumerated) given a pointer to any object
- Objects have names
- Objects have relationships to each other – parent, sibling(s), children
- Objects have behavior

DataPump Object System

- DataPump objects are used to collect and represent all the major data structures within the DataPump
- Objects have names: typically the names look like normal DNS names (e.g., "msc.fn.mcci.com")
  - Names MCCI creates always end in ".mcci.com"
  - Customers can do what they like
  - Multiple objects might have identical names, but can be distinguished by their instance numbers.
  - Fully qualified name in a printout is "msc.fn.mcci.com#0"
- We have library routines that can enumerate all objects using a pattern for the name; with limited wild cards:
  - e.g., you can browse for all objects matching "*.fn.*"
  - This makes it easy for a client to match all objects of a given "kind"
    - provided that names follow predictable patterns

Object System (2)

- Objects have behavior
  - All objects can receive “IOCTL” operations
  - IOCTLs always have a common stereotype:
    - IoctlFn(pObject, IoctlCode, pInArg, pOutArg)
  - An object may choose to claim an IOCTL or may choose not to claim it
    - if it doesn’t claim the IOCTL, the DataPump will try to send the IOCTL to the next logical recipient
  - IOCTL codes directly represent the size of the in and out arguments (as part of the numerical code)
  - When an object is created, the creator specifies who the “next logical recipient” for IOCTLs should be. This next recipient is called the “IOCTL parent”
  - Because of all these fine points, IOCTLs are routable by the IOCTL system in the core DataPump, and you can get inherited behavior
  - If an object doesn’t supply a behavior, it’s IOCTL parent will be asked to provide a behavior, so the child object can inherit from its parent
- We use this at MCCI to modularize the code and allow very high levels to tunnel through to very low levels

Object System (IOCTLs)

- Every UPLATFORM is an object; it's a IOCTL child of the root object
- Every USBPUMP_HCD is an object; a IOCTL descendent of its UPLATFORM (via USBPUMP_PHY).
- USBPUMP_USBD is an object; an IOCTL descendent of the UPLATFORM. (A USBD can manage multiple HCDs, so it can't pass IOCTLs to any particular HCD).
- Each class driver is an object, and an IOCTL child of the USBD that owns the class driver.
- Each class driver device instance (USBPUMP_USBDI_FUNCTION) is an object. When attached to a USBPUMP_USBDI_PORT, the port is the IOCTL parent of the function.
- Each USBPUMP_USBDI_PORT is an object, representing a physical attach point of a device. Its IOCTL parent is the USBPUMP_USBDI_FUNCTION of the parent hub.
- A client of a USBPUMP_USBDI_FUNCTION can send an IOCTL to its function instance, and by inheritance, that IOCTL will flow down to the object (possibly the UPLATFORM) that provides the implementation.
- Since UPLATFORM behavior is determined on a platform-by-platform basis, this is one of the primary ways to customize the run-time behavior of the DataPump for a specific OEM requirements
- If a behavior isn't implemented, then an appropriate error code is returned to the issuer of the IOCTL

Object System (4)

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

Object System (5)

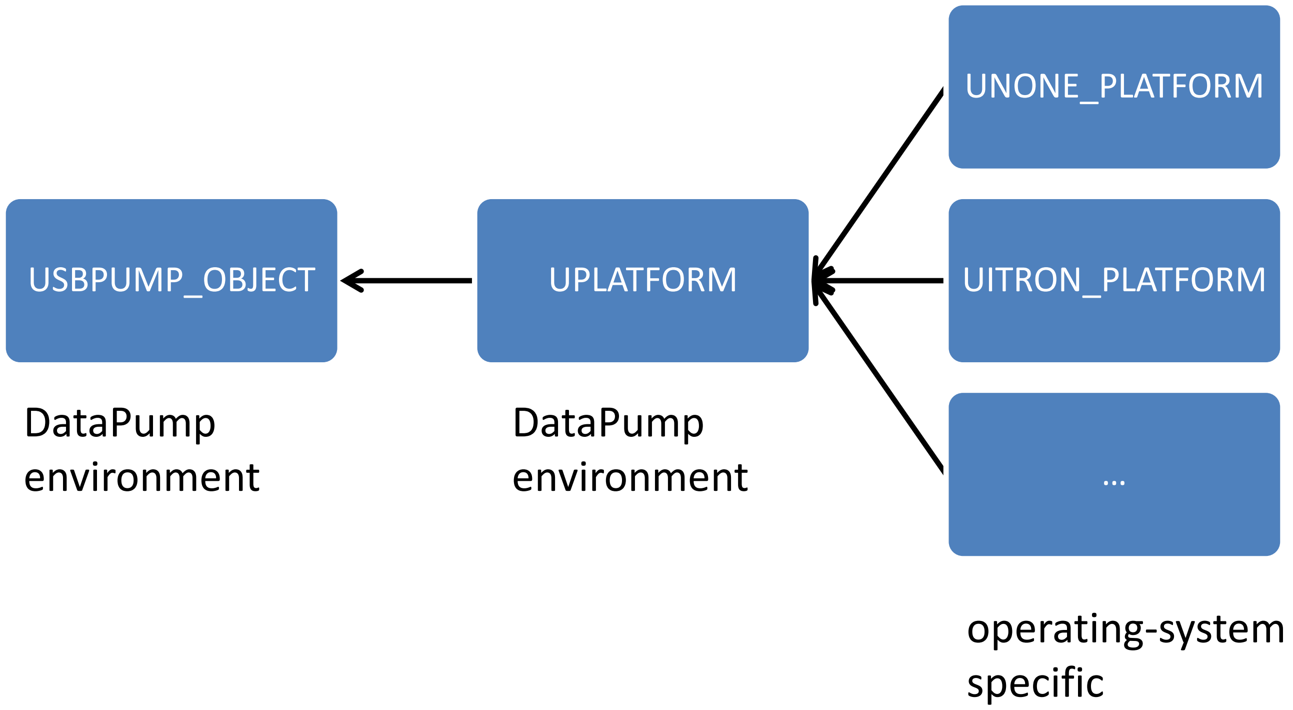
- Object System was initially designed for the Host stack, but was first used in the Device stack.
- There are additional features that are only used in the host stack
  - The object system is used to collect & manage instances of (host) Class Drivers; this requires mechanisms that are not used for those objects that are part of the device stack
  - However, the unification allows for clients in OTG to connect to transport objects without worrying about whether the transport object is a “device-stack” object or a “host-stack” object
    - consider, for example, two OTG Ethernet devices (USBPUMP_ABSTRACT_NIC) or two OTG MTP devices…

UPLATFORM

UPLATFORM basics

- Represents the underlying operating system and target board to the DataPump
- Every UPLATFORM is a DataPump Object
- Normally, only one per DataPump task
- Normally, the concrete instance for a given platform is derived from UPLATFORM

UPLATFORM type derivation diagram



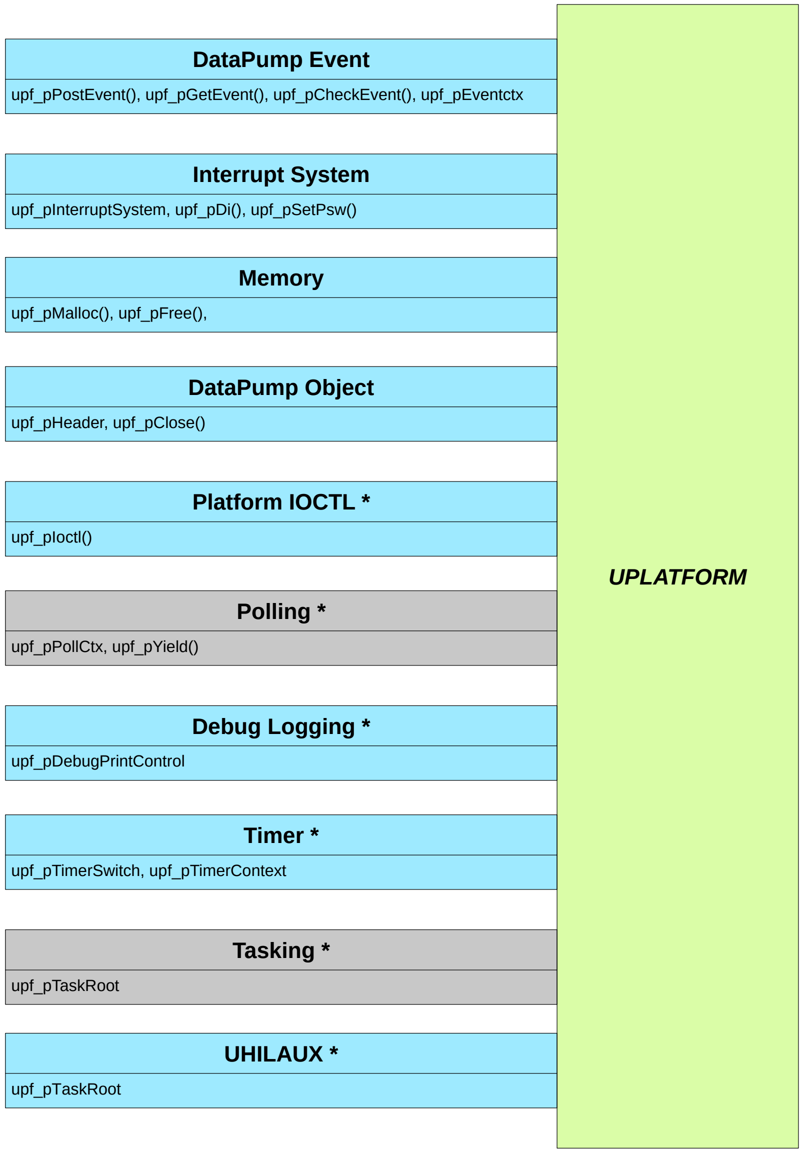
Platform Methods

- The basic binding of the DataPump to the actual code on the platform is done using pointers to functions
- These functions are accessed via the *platform pointer*, which is a pointer to a UPLATFORM object
- The UPLATFORM object is normally an abstract view of a concrete object that varies on a platform-by-platform basis
- Because the function pointers are conceptually like C++ class methods (if the UPLATFORM were a C++ class), we call them “platform methods”.
- The “core DataPump” has no idea how the platform methods are actually implemented; the collection of platform methods for a given set of operations form an *interface* (in the UML sense) that is exported by the concrete platform. (In a C++ sense, these are like “virtual methods”)
- In the current version, there is no explicit grouping of methods into UML-interfaces, but we may introduce this in the future. Following slides show the logical grouping.
  - “interface” here is used in the UML sense, not USB; like Objective-C “protocols”.

UML View of the Platform

- UML-Interfaces:
  - **DataPump Object**: upf_Header, upf_pClose()
  - **DataPump Event**: upf_pPostEvent(), upf_pGetEvent(), upf_pCheckEvent(), upf_pEventctx – this mandatory interface provides the abstract DataPump Event API
  - **Interrupt System**: upf_pInterruptSystem, upf_pDi(), upf_pSetPsw() – this mandatory interface provides an abstract interrupt system
  - **Memory**: upf_pMalloc(), upf_pFree() – allocate and free memory
  - **Platform IOCTL**: upf_pPlatformIoctl() – this is optionally provided to do filtering for platform-specific IOCTLs.
  - **Polling**: upf_pPollCtx, upf_pYield() – this optional interface provide some additional control in non-preemptive systems.
  - **Debug Logging**: upf_pDebugPrintControl – this optional interface allows messages to be displayed
  - **Timer**: upf_pTimerSwitch, upf_pTimerContext –- this interface is used by the host stack (but generally not by the device stack); it provides an millisecond timer service.
  - **Tasking**: upf_pTaskRoot – this optional interface provides inter-task communication. Not used by the host stack.
  - **UHILAUX**: this optional, but very important, interface provides DMA abstraction services. Very important for host applications of the DataPump.

UML Interface View: UPLATFORM


DataPump Event
upf_pPostEvent(), upf_pGetEvent(), upf_pCheckEvent(), upf_pEventctx
Interrupt System
upf_pInterruptSystem, upf_pDi(), upf_pSetPsw()
Memory
upf_pMalloc(), upf_pFree(),
DataPump Object
upf_pHeader, upf_pClose()
Platform IOCTL *
upf_pIoctl()
Polling *
upf_pPollCtx, upf_pYield()
Debug Logging *
upf_pDebugPrintControl
Timer *
upf_pTimerSwitch, upf_pTimerContext
Tasking *
upf_pTaskRoot
UHILAUX *
upf_pTaskRoot
UPLATFORM

DATAPUMP EVENT HANDLING

DataPump Event Handling Background

- The base MCCI USB DataPump is implemented as a single task
- This task has (conceptually) a very simple structure:

```
Initialize();
while (TRUE)
 {
 CALLBACKCOMPLETION * CONST e =
      WaitForNextEvent();
 DispatchEvent(e);
 }
```

- For historical reasons, the names are different. For handling unusual applications, the code looks more complicated
- But this is the basic logic of the DataPump.

More about events

- Events are dispatched strictly sequentially (FIFO order)
- Events are actually pointers to CALLBACKCOMPLETION structures.
- CALLBACKCOMPLETION structures have two fields:
  - callback_pfn: pointer to a function to be called
  - callback_ctx: arbitrary (VOID*) pointer to be used by the called function.
- The callback function has prototype

```
VOID (*pfn)(
    VOID *ctx
    );
```

- This is standard event-driven programming style.
  - We wish that we'd added more arguments to the callback function; but it's too late to change now.

Posting Events

- Events are posted to the DataPump task three ways:
  - from inside the DataPump task
  - from other tasks running concurrently with the DataPump task
  - from ISRs
- We post from “inside” mainly to limit how deep the stack gets, or to defer processing of lengthy operations while we do low-latency processing.
  - If we post from inside, the event will not be processed until all previous events have been fully processed.
- We post from other tasks as the primary way of synchronizing from the DataPump clients to the DataPump
  - Exact mechanisms are platform specific but result is the same – a call back function is called in DataPump context via the event loop.

Posting Events (2)

- ISRs for chip drivers are normally written in a platform independent way
- ISRs normally can't access most chip resources
- We post events from ISRs in order to ensure that we are not interrupting the DataPump and to allow processing to continue within the DataPump task
- Chip drivers ISRs assume shared address space with DataPump task.
- Sometimes the posting of events is hidden by other APIs, specifically:
  - `UHIL_SynchronizeToDataPumpDevice()`
  - `UsbPumpPlatform_SynchronizeToDataPump()`

Posting Events (3)

- The event-queue interface exported by the platform consists of three platform methods:
  - upf_pPostEvent()
  - upf_pGetEvent()
  - upf_pCheckEvent()
- The core DataPump implements all the other APIs in terms of these methods.
- Unless specifically permitted by the platform, these methods can only be called within the DataPump task or from ISRs
- The core DataPump also references an UEVENTCONTEXT *

UEVENTCONTEXT

- Abstract handle used for the Platform Event interface
- Though it's confusing to figure out, this is never dereferenced by the *core* DataPump
- It's defined as a structure type
- The concrete platform must define how this pointer is used and the contents if any of the underlying structure
- A default layout is provided in "uhilevent.h" but the platform is not required to use this!
  - This layout is used by the "default" event handling functions that are part of the DataPump library, and by some platforms.
- The value comes from upf_pEventctx, and is set up by platform initialization.

Events (4)

```
VOID pPlatform->upf_pPostEvent(
        UEVENTCONTEXT *pCtx,
        CALLBACKCOMPLETION *pCompletionBlock
        );
```

- This function is supposed to schedule pCompletionBlock to be processed later
- Multiple calls using the same pCompletionBlock pointer should result in multiple calls to the callback_pfn.
- This may be implemented by having a ring buffer of pointers to completion blocks, and just put pCompletionBlock into that ring buffer
- The *default* implementation uses a fixed-size ring buffer
- There is no requirement to use a ring buffer!!!!
- If the buffer gets full, it’s a system failure

Events (5)

- Windows uses the native message queue instead of using a ring buffer
- OSE uses the native signal queue for messages from outside, but uses a ring buffer for messages from the inside and for an optimization sending from the same address space
  - Only sends a signal() if the buffer is empty
- Nucleus uses a native message queue allocated for the DataPump task during initialization

Events (6)

- The native OS message queue can be used if the native OS message implementation is efficient
- A ring buffer + the native OS message queue should be used if the native message implementation is slow
- A ring buffer + some other OS primitive must be used if the OS doesn’t provide a simple message queuing operation
- The default implementation can be very convenient for these hybrid approaches – see OSE for an example of how to do it

Events (7)

- The basic idea is as follows:
  - Get all events from the ring buffer
  - Maintain the invariant that the DataPump task blocks on the native API only if the ring buffer is empty
  - Therefore when the ring buffer transitions from empty to not-empty, we must assume that the DataPump task might be blocked; so we must use the native “post” API to unblock it
  - If the ring buffer is not empty, we know that the DataPump is unblocked, so we don’t use the native API, we just add the message.

UEVENTNODEs

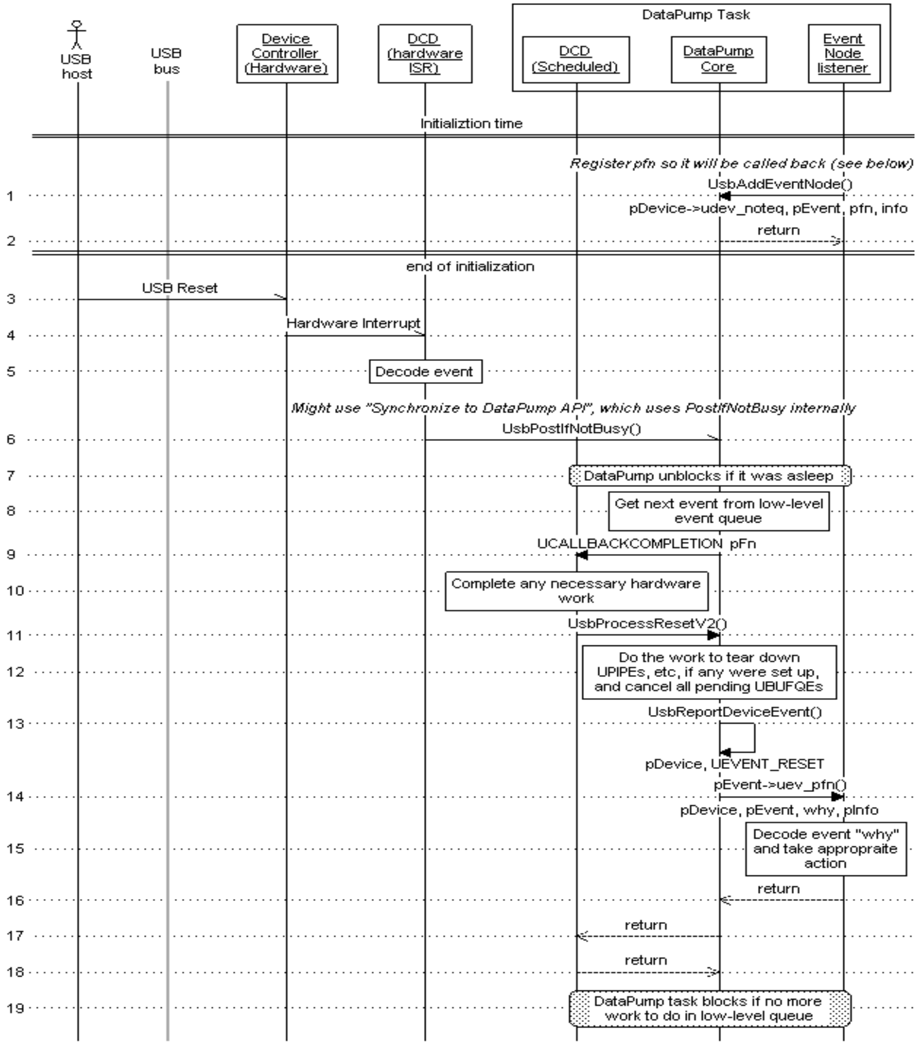
- Represent callback attach points
- Contain pointer to function, pointer to context (standard “continuation” or “closure” information)
- Also contain linkage fields, so that the DataPump can put them into a linked list
- The client chooses the events of interest by choosing the list head pointer
- UDEVICE, UCONFIG, UINTERFACESET, UINTERFACE, UPIPE objects all have UEVENTNODE list headers

UEVENTNODE (2)

- See UEVENTNODE, UEVENTFN and UEVENT in the .chm file for low-level details
- These are always processed within DataPump context
  - in cell phones, DataPump is a task, so this means these are called from within the DataPump task
- The hardware ISR uses a different mechanism to send events to the DataPump; the DataPump calls **`UsbReportEvent()`** to actually report an event to a chain of UEVENTNODEs

Examples of UEVENTNODE Processing

- See training-event-sequence-1.sds or .pdf


DataPump Task
USB host
USB bus
Device Controller (Hardware)
DCD (hardware ISR)
DCD (Scheduled)
DataPump Core
Event Node listener
Initializtion time
Register pfn so it will be called back (see below)
1
UsbAddEventNode()
pDevice->udev_noteq, pEvent, pfn, info
return
2
end of initialization
3
USB Reset
4
Hardware Interrupt
5
Decode event
Might use "Synchronize to DataPump API", which uses PostIfNotBusy internally
6
UsbPostIfNotBusy()
7
DataPump unblocks if it was asleep
8
Get next event from low-level event queue
9
UCALLBACKCOMPLETION pFn
10
Complete any necessary hardware work
11
UsbProcessResetV2()
12
Do the work to tear down UPIPEs, etc, if any were set up, and cancel all pending UBUFQEs
13
UsbReportDeviceEvent()
pDevice, UEVENT_RESET
14
pEvent->uev_pfn()
pDevice, pEvent, why, pInfo
15
Decode event "why" and take appropraite action
16
return
17
return
18
return
19
DataPump task blocks if no more work to do in low-level queue

DEBUG LOGGING SYSTEM

Debug Logging System

- The UPLATFORM structure contains pointers to low-level debug output routines
- These routines are accessed via the macros TTUSB_PRINTF(), TTUSB_LOGF(), and TTUSB_FLUSH().
  - Except for FLUSH(), these can safely be called from ISRs
  - FLUSH() should be used sparingly, because it blocks the DataPump until all output has been drained
- Normally, MCCI implements these routines as a ring buffer plus an asynchronous printing function. This minimizes the performance impact.

Debug Log Macros

- TTUSB_PLATFORM_PRINTF(), TTUSB_OBJECT_PRINTF() are only expanded in “checked” builds of the firmware.
  - So the “free” build is smaller and faster
- Synopsis:

  -
    ```
    TTUSB_PLATFORM_PRINT((
      UPLATFORM *pPlatform,
      UINT debugPrintFlags,              // see next slide
      CONST CHAR *pFormat, …
      ));
    ```
- Note the double (( )) – this allows us to deal with a variable-length argument list on any compiler.
- We don’t use computed formats; the format string is available at compile time. This eases integration with other debug/tracing systems.

Debug Log Flags

- The debug log flags indicate the *kind of message* that is being produced.
- A message may be tagged as being of several kinds, but this is not recommended for new code.
- Two special flag values **must not** be combined with any other value
  - UDMASK_FATAL_ERROR marks fatal errors; normally this is the last message output before a system abort.
  - The special value 0 indicates that the message should be output unconditionally (but this may involve different treatment from a fatal error).

Debug Log Flags (1)

Name	Message is …	Volume of output
UDMASK_ERRORS	An "error" message	Low
UDMASK_ANY	An "informational" message	Low after initialization; moderate during init.
UDMASK_FLOW	A detailed flow message	High
UDMASK_CHAP9	Related to chapter 9 events	Low
UDMASK_PROTO	(Device stack only)	Moderate
UDMASK_BUFQE	(Device stack only)	High
UDMASK_HWINT	Related to hardware interrupts	High
UDMASK_TXDATA	Related to transmit data (host: OUT pipes)	High
UDMASK_RXDATA	Related to receive data (host: IN pipes)	High

Debug Log Flags (2)

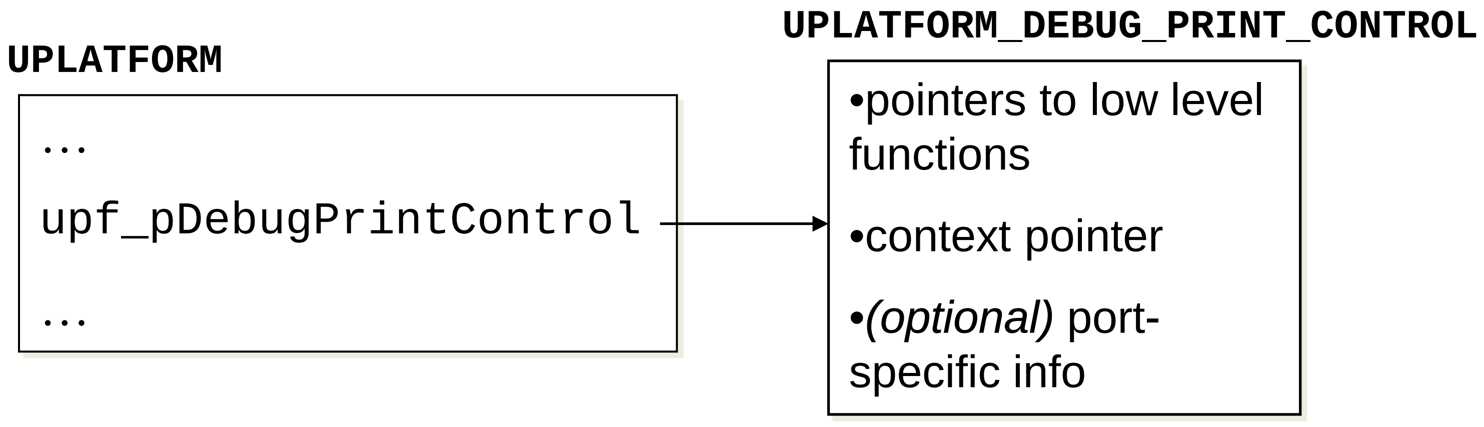
Name	Message is …	Volume of output
UDMASK_HWDIAG	Related to hardware diagnostics	High
UDMASK_HWEVENT	Related to hardware events	High
UDMASK_VSP	(device stack only)	Moderate
UDMASK_ENTRY	Related to function entry/exit	High
UDMASK_ROOTHUB	Related to the root hub	Low
UDMASK_USBDI	Related to USBDI	Moderate
UDMASK_HUB	Related to hub operation	Low except on insertion/removal
UDMASK_HCD	Related to an HCD	High

Debug Log Primitives

- The logging primitives can be used in checked or free builds, but because they unconditionally pull the debug output routines into the link, they make the free build larger.
  - UsbPumpPlatform_Printf() outputs a message to the debug log based on debug flags in the UPLATFORM.
  - UsbPumpObject_Printf() outputs to the debug log, based on debug flags set in a given USBPUMP_OBJECT_HEADER.
  - UsbPumpPlatform_Flush() flushes a UPLATFORM debug log.
- These use the formatting functions from the USB common library.

Low Level Debug Logging

- Architecture:


UPLATFORM
…
upf_pDebugPrintControl
…
UPLATFORM_DEBUG_PRINT_CONTROL
•pointers to low level functions
•context pointer
•(optional) port-specific info

➢ The entire print-control structure is optional, and may be omitted on ports for which debug logging makes no sense

Debug Print Initialization

- This must be done during Platform/OS initialization.
  1. Allocate the `UPLATFORM_DEBUG_PRINT_CONTROL` structure in read/write memory.
  2. Use the macro to initialize the structure: `UPLATFORM_DEBUG_PRINT_CONTROL_RUNTIME_INIT_V1()`
  3. Then inform the DataPump:

```
/* Initialize print handler */
UHIL_PrintInit(pPlatform, pDebugStructure);
UHIL_DebugPrintEnable(pPlatform, TRUE);
```

Debug Print Control Contents

- Functions: pPrintBuf, pPrintBufTransparent, pFlush, pPrintPoll, pDebugPrintEnable, pClose
- Data: pContext
- Additional User Data

Debug Print Control: pPrintBuf

- Prototype:

```
VOID (*pPrintBuf)(
  UPLATFORM *pPlatform,
  CONST CHAR *pBuffer,
  BYTES nBytes
  );
```

- Prints nBytes starting at pBuffer
- Input buffer follows UNIX semantics (i.e., \n is <crlf>).
- Translates \n to \r\n if needed by lower level; otherwise same as pPrintBufTransparent.

Debug Print Control: pPrintBufTransparent

- Prototype:

```
VOID (*pPrintBufTransparent)(
  UPLATFORM *pPlatform,
  CONST CHAR *pBuffer,
  BYTES nBytes
  );
```

- Prints nBytes starting at pBuffer, outputting exactly as given without any formatting.
- Depending on printing environment, might be identical to pPrintBuf.

Debug Print Control: pFlush

- Prototype:
```
VOID (*pFlush)(
  UPLATFORM *pPlatform
  );
```
- Delays execution until all queued debug output (from previous pPrintBuf operations) has been output.
- Depending on printing environment, might do nothing.
- May be NULL if no flush operation is needed on this platform.

Debug Print Control: pPrintPoll

- Prototype:

```
VOID (*pPrintPoll)(
  UPLATFORM *pPlatform
  );
```

- Helper function on non-interrupt driven platforms
- Called periodically by DataPump
- Typical use: if any characters are in the ring buffer, push them to the UART fifo until the UART fifo is full; otherwise do nothing.
- May be NULL if not needed on this platform.

Debug Print Control: pDebugPrintEnable

- Prototype:

```
BOOL (*pDebugPrintEnable)(
  UPLATFORM *pPlatform,
  BOOL fEnableIfTrue
  );
```

- Enable/Disable debug output.
- Provided as a convenience routine for use by higher level code.
- Platform initialization always enables
- Result is previous state (TRUE if was enabled, FALSE if was disabled)

Debug Print Control: pClose

- Prototype:

```
VOID (*pClose)(
  UPLATFORM *pPlatform
  );
```

- Called during DataPump shutdown
- Should close any device handles, and release any memory used by the debug print context.
- May be NULL if not needed on this platform.

If you have to do a port….

- To port the DataPump to a new platform, refer to 950000260c (the DataPump Platform Porting Guide)
- This is currently at revision C and documents V3 of the DataPump

SETTING UP MCCI'S USBD – INITIALIZATION APIS

How initialization works in detail

- Initialization of the Host Stack is table-driven
- Table interpreter is UsbPump_GenericHostInit()
  - Processes an application init vector (a collection of nodes which specify how the components should be initialized)
  - When using the normal DataPump init code, you’ll call this from the device “init-finish” function.
  - When using the new os/none simplified initialization, you’ll write a function named “UsbPump_ApplicationInit()”, and you’ll call UsbPump_GenericHostInit() from your application init function.

More on Initialization

- UsbPump_GenericHostInit() initializes one (or more) instances of USBD, each USBD managing a collection of HCDs
  - USBPUMP_HOST_INIT_NODE_VECTOR is the top-level structure, which points to an array of USBPUMP_HOST_INIT_NODEs.
- It *conditionally* skips creation of a given USBD, based on the result of calling a probe function given in the USBPUMP_HOST_INIT_NODE.
- Each USBPUMP_HOST_INIT_NODE controls one USBD instance, along with its host contollers
- For throughput and simplicity, one USBD per HCD is best – but has a larger data RAM footprint
- Example on next page.

Explicitly modeling the PHY: USBPUMP_PHY

- This is an advanced topic, but …
- If you're doing OTG, or combining USB with non-USB things, we model the PHY with an extra object
  - Example: the Type C connector DisplayPort mode
  - Example: SSIC USB which has explicit M-PHY behavior)
  - Example: dual-role devices
- In this case, the PHY must be created before creating the host stack (or device stack).
- For simple host stacks, the PHY is not required.
- Some of our examples always go through the motion of creating PHYs anyway, using UsbPump_GenericPhyInit(). This is so our sample code will work unchanged on OTG controllers.

The Host Init Node Vector

```
static
CONST USBPUMP_HOST_INIT_NODE sk_HostInitNodes[] =
        {
        USBPUMP_HOST_INIT_NODE_INIT_V3(                                 \
                /* pProbeFn */ NULL,                                    \
                /* pUsbdInitFn */ UsbPumpUsbd_Initialize,               \
                /* pUsbdConfig */ &sk_UsbPumpUsbd_Config,  7            \
                /* pUsbdImplementation */                               \
                        &gk_UsbPumpUsbdiUsbdImplementation_Minimal, 4   \
                /* pUsbdTTInitFn */ NULL,                               \
                /* pUsbdTTConfig */ NULL,      5                        \
                /* pWirelessUsbdInitFn */ NULL,                         \
                /* pWirelessUsbdConfig */ NULL,                         \
                /* pSuperSpeedInitFn */ UsbPumpUsbdSuperSpeed_Initialize, \ 6
                /* pSuperSpeedConfig */ &sk_UsbPumpUsbdSuperSpeed_Config, \
             8 /* DebugFlags*/ 0,                                       \
                /* fDoNotStartUsbd)*/ FALSE,                            \
                /* pHcdInitVector */ &gk_UsbPumpHcd_GenericInitVector,  \ 2
                /* pDriverClassInitVector */ &sk_ClassDriverInitHeader  \
                )                                                       3
        };

static
CONST USBPUMP_HOST_INIT_NODE_VECTOR sk_UsbPumpHost_GenericInitVector =
        USBPUMP_HOST_INIT_NODE_VECTOR_INIT_V1(
                /* link to the vector */ sk_HostInitNodes,  1
                /* prefunction */ NULL,
                /* postfunction */ HostMscApp_Host_InitFinish  9
                );
```

Notes on Host Init Node Vector

(1) The `USBPUMP_HOST_INIT_NODE_VECTOR` points to the array of `USBPUMP_HOST_INIT_NODE`s; the macro calculates the size of the array.

(2) Each `USBPUMP_HOST_INIT_NODE` points to a similar structure that describes all HCDs that are governed by this USBD instance. (See "HCD Configuration" below for more details)

(3) The node points to a similar structure that describes all class drivers (and the matching algorithms) for this USBD instance

(4) The UsbdImplementation supplies dispatch tables. A number of different implementation tables are provided by MCCI, and this controls how much code is needed for USBD. You must use one of the MCCI pre-supplied tables.

(5) If bus bandwidth scheduling is needed for transaction translators, you should supply the pointer to the MCCI TT init function, and a configuration structure. This isn't needed for simple enumeration.

(6) This pair of values configures this USBD for SuperSpeed support. The configuration structure provides defaults, and the function (which is always `UsbPumpUsbdSuperSpeed_Initialize`) is used to cause the linker to pull in the appropriate support modules. Set these to NULL if your host controller doesn't support SuperSpeed.

(7) The USBD config structure provides additional configuration info to the USBD about required capabilities. This influences RAM footprint.

(8) The debug flags provide default values that are OR'ed into the object debug masks for create objects, indicating the kinds of messages that should be output to the log.

(9) The postfunction is called after processing all the . Typically this can be used to set up local operating system bindings, etc. However, this can also be done after all initialization is completed (when UsbPump_GenericHostInit() returns).

The USBD configuration structure

```
static
CONST USBPUMP_USBDI_USBD_CONFIG sk_UsbPumpUsbd_Config =
        USBPUMP_USBDI_USBD_CONFIG_INIT_V9(
                /* name */ NULL,                  1
              2 /* maxNestedCompletions */ 0,
                /* sizeConfigBuffer */ 512,         3
                /* sizeStringDescBuffer */ 128,
                /* maxUrbExtraBytes */ 0,
                /* attach debounce */ USBPUMP_USB20_TATTDB_DEFAULT,
                /* reset recovery */ USBPUMP_USB20_TRSTRCY_DEFAULT,
                /* set address completion */ USBPUMP_USB20_TDSETADDR_DEFAULT,
                /* set address recovery */ USBPUMP_USB20_TSETADDRRCY_DEFAULT,     4
                /* std request no data */ USBPUMP_USB20_TDRQCMPLTND_DEFAULT,
                /* std request data1 */ USBPUMP_USB20_TDRETDATA1_DEFAULT,
                /* std request last data */ USBPUMP_USB20_TDRETDATAN_DEFAULT,
                /* a_tStdRequestMinimum */ USBPUMP_USB20_TSTDREQUEST_MINIMUM_DEFAULT,
                /* resume recy */ USBPUMP_USB20_TRSMRCY_DEFAULT,
                /* number of hubs */ 0,        5
                /* max ports/hub */ 7,
                /* a_tHostInitiatedResumeDuration */
                                USBPUMP_USB20_LPM_HIRD_DEFAULT,  4
                /* hub ID overrides */ NULL,
                /* annunciator sessions */ 0,    6
                /* debug */ UDMASK_CHAP9 | UDMASK_USBDI | UDMASK_HUB,  7
              8 /* LPM idle */ 0,
                /* "flags" */ 0,   9
                /* bMinU1Timeout */ 0,
                /* bMinU2Timeout */ 0,
                /* bHubU1Timeout */ 0,       10
                /* bHubU2Timeout */ 0
                );
```

Notes on USBD Configuration

(1) The name of the USBD object can be overridden if needed.

(2) maxNestedCompletions is used to limit stack depth (at cost of CPU overhead). Zero means never nest completions

(3) sizeConfigBuffer must be set as large as the largest config bundle you plan to support. It must be at least 512. sizeStringDescBuffer must be set as large as the largest serial number descriptor you need to distinguish. It's given in bytes.

(4) These configuration constants allow you to adjust the timing used by USBD. The names match the spec. Note that tStdRequestMinimum sets a minimum timeout – the timeout will never be less. We recommend use of 100 (meaning 100ms).

(5) Very important, this controls how many external hubs will be supported by this instance of USBD. (It's the cumulative total across all HCDs. In this case we wrote zero, so external hubs are not supported.) The hub driver is always instantiated with enough instances to cover all the root hubs.

(6) This controls the number of annunciator sessions that will be supported. Annunciator sessions are used for announcing USB system changes to external software. If there are no clients, you can set this to zero.

(7) Debug flags for USBD and hub objects.

(8) Set to number of microseconds prior to requesting LPM. This is used for both software and hardware LPM. See config flags.

(9) Configuration "flags" allow certainly optional features to be enabled. See next slides.

(10) The U1/U2 timeout constants are for SuperSpeed only; if left zero, safe defaults are used by the host stack.

USBD Configuration Flags (intro)

- These flags control the operation of USBD in a number of obscure or corner cases
- They're mostly related to power management
- One flag controls serial number filtering
- Recommendation: during system bring-up, set all flags to zero.
  - System architects should carefully review flags and determine whether they're needed.
  - Some flags may be required if you want to get USB-IF embedded host certification (especially for silicon testing). Systems have more leeway for certification than silicon does.

USBD Configuration Flags (1)

Flag (USBPUMP_USBDI_USBD_CONFIG_FLAG…)	Means…
ENABLE_AUTO_LPM	Enable automatic (software) LPM. If not set, LPM is only supported if host controller supports it and HW enable flag is set. (XHCI 1.0 does.)
ENABLE_U1	Enable switching to U1. This saves power, but older USB3 devices may have problems. SuperSpeed only.
ENABLE_U2	Enable switching to U2. See ENABLE_U1. SuperSpeed only.
ENABLE_LTM	Allows the stack to use LTM. Stack will enable LTM messaging from devices that report LTM support. SuperSpeed only.

USBD Configuration Flags (2)

Flag (USBPUMP_USBDI_USBD_CONFIG_FLAG…)	**Means…**
DONT_VALIDATE_SERIAL_NUM	If set, then any string from the device will be accepted as a serial number (even if it is not legal by Windows standards). If reset, a serial number that would be rejected by Windows is also rejected by the stack. (Serial number chars must be in 0x20..0x7F and not equal to 0x2C.) Device will still be accepted, but USBD will pretend it has no serial number
WAKE_ON_CONNECTION	Set up hubs to wake the host from suspend if a device is connected.
WAKE_ON_DISCONNECTION	Set up hubs to wake the host from suspend if a device is disconnected.
WAKE_ON_OVERCURRENT	Set up hub ports to wake the host from suspend if overcurrent is detected.

USBD Configuration Flags (3)

Flag (USBPUMP_USBDI_USBD_CONFIG_FLAG…)	Means…
ENABLE_HUB_AUTO_SUSPEND	If no device is connected, suspend the hub (will wake up on device connect)
PORT_POWER_OFF_WHEN_SLEEP	Power off external ports when system is going to sleep
PORT_POWER_OFF_WHEN_RESTART	Power off external ports when system is restarted
PORT_POWER_OFF_WHEN_SHUTDOWN	Power off external ports during controlled system-shutdown.
ENABLE_HARDWARE_LPM	Enable hardware-controlled automatic LPM (XHCI 1.0 supports this)
HARDWARE_LPM_MODE_1	Enable hardware LPM mode 1 instead of mode 0. See XHCI 1.1 section 4.23.5.1.1.1. Only relevant if XHCI version is 1.1 or later.
DONT_LPM_IF_MISMATCH_BESL	Disable LPM if host supports BESL and device does not; or if device supports BESL but hub does not.

HCD Configuration

- TrueTask USB supports building one image that handles multiple different HCDs, possibly running on different hardware
- HCDs are initialized by parsing an USBPUMP_HOST_HCD_INIT_NODE_VECTOR, which specifies the possible host controllers
  - Each USBPUMP_HOST_INIT_NODE contains a pointer to an instance of this structure.
- Simplest case: one HCD.
- Each entry is one USBPUM_HOST_HCD_INIT_NODE, which describes a possible host controller
  - Logic is provided to “probe” (test for hardware presence) prior to creating the HCD.
- DataPump sample code uses an USBPUMP_HOST_HCD_INIT_NODE_VECTOR provided by the platform layer.
- For user code, it is convenient to specify the HCD initialization in the same file as your USBPUMP_HOST_INIT_NODE
- Normally you must supply setup info that is specific to your HCD, in addition to the setup info required by the DataPump.

HCD Configuration Details (FTDI FT900 EHCI)

```
/*
|| This table provides the initialization information for the HCDs
*/
static
CONST
USBPUMP_HOST_HCD_INIT_NODE
sk_Ft900_HcdInitNodes[] =
        {
        USBPUMP_HOST_HCD_INIT_NODE_INIT_V1(
                /* pProbeFn */          NULL,
                /* pInitFn */           ehcihcd_Init,  ②
                /* pConfig */           &sk_Ft900_EhciConfig,  ③
                /* DebugFlags */        0
                )
        };

CONST
USBPUMP_HOST_HCD_INIT_NODE_VECTOR
gk_UsbPumpHcd_GenericInitVector =
        USBPUMP_HOST_HCD_INIT_NODE_VECTOR_INIT_V1(
                /* name of the vector */sk_Ft900_HcdInitNodes,  ①
                /* prefunction */       NULL,
                /* postfunction */      NULL
                );
```

(1) As with the host vector, the HCD init vector macro needs the name of the list of init nodes. It automatically calculates the length of the vector.

(2) You must supply the name of the HCD initialization function. This can be found in the HCD documentation. For the FT900 EHCI variant, the function is ehcihcd_Init.

(3) For the FT900 EHCI variant, you must supply a pointer to an HCD-specific configuration structure. We show this on the next page.

HCD Configuration Details (FTDI FT900 EHCI)

```
/*
|| HCD configuration
*/
static
CONST
EHCIHCD_CONFIG_INFO
sk_Ft900_EhciConfig =
        EHCIHCD_CONFIG_INFO_INIT_V5(
                /* hBus */              (UHIL_BUSHANDLE) 0,          ①
                /* IoPort */            (IOPORT) FTDI_FT900_EHCI,
                /* hUsbInt */
                        OSNONE_FT900_INTERRUPT_RESOURCE_HANDLE_FROM_IRQ(  ②
                                FTDI_FT900_IRQ_USB_HOST
                                ),
                /* bNakCountReload */   EHCIHCD_NAKCOUNT_RELOAD_DEFAULT,      ③
                /* bIntrThreshold */    EHCIHCD_INTERRUPT_THRESHOLD_DEFAULT,  ④
                /* bIntrDelayInFrame */ FT900_EHCI_INTERRUPT_DELAY_IN_FRAMES,
                /* pGetBaseAddrFn */    ft900Ehci_GetBaseAddress,                 ⑤
                /* ulFlags */           EHCIHCD_CONFIG_FLAG_USE_NFRAMELIST_256,  ⑥
                /* size */              sizeof(USBPUMP_HCD_FT900)  ⑦
                );
```

(1) Bus handle and I/O port are system specific. Normally bus handle is NULL.
(2) hUsbInt specifies the interrupt connection. It's specified as an Interrupt Resource Handle; these are OS and Platform Specific.
(3) bNakCountReload, bIntrThreshold, program standard parameters in the EHCI register set
(4) bIntrDelayInFrame is an estimate of how long it takes for interrupts to be delivered to the HCD
(5) pGetBaseAddrFn is used by the common EHCI code to find the base address on systems for which the address is not fixed in advance. For the FT900, use this function.
(6) USE_NFRAMELIST_256 causes the EHCI driver to use a 256-entry frame list instead of the normal 1K frame list, if supported by hardware. This saves RAM
(7) This specifies the overall size of the HCD instance object. Use this for the FT900 HCD.

HCD Configuration Details (XHCI)

```
static CONST USBPUMP_HOST_HCD_INIT_NODE sk_Cameo_HcdInitNodes[] =
        {
        USBPUMP_HOST_HCD_INIT_NODE_INIT_V1(
                /* pProbeFn */          NULL,
                /* pInitFn */           UsbPumpCameo_XHcdInit, ②
                /* pConfig */           &sk_Cameo_XhcdConfig,
                /* DebugFlags */        0                     ③
                )
        };

CONST USBPUMP_HOST_HCD_INIT_NODE_VECTOR gk_UsbPumpHcd_GenericInitVector =
        USBPUMP_HOST_HCD_INIT_NODE_VECTOR_INIT_V1(
                /* name of the vector */sk_Cameo_HcdInitNodes, ①
                /* prefunction */       NULL,
                /* postfunction */      NULL
                );
```

(1) As with the host vector, the HCD init vector macro needs the name of the list of init nodes. It automatically calculates the length of the vector.

(2) You must supply the name of the HCD initialization function. This can be found in the HCD documentation. For the uITRON variant, the function is UsbPumpCameo_XHcdInit.

(3) For all XHCI variants, you must supply a pointer to an XHCD-specific configuration structure. We show this on the next page.

HCD Configuration Details (XHCI)

```
static CONST USBPUMP_XHCD_CONFIG_INFO sk_Cameo_XhcdConfig =
        USBPUMP_XHCD_CONFIG_INFO_INIT_V9(                  1
        /* ulWiring */            0,
        /* hBus */                (UHIL_BUSHANDLE) 0,      2
        /* IoPort */              (IOPORT) CAMEO_XHCI_REG_BASE,
        /* hUsbInt */             USBPUMP_UITRON_MAKE_INTERRUPT_RESOURCE_HANDLE( 3
                                          g_Cameo_UsbInterruptResource
                                          ),
        /* MaxStreamId */                 64,
        /* MaxSlots */                    64,
        /* nBufSeg */                     2,
        /* SizeOfEventRing */             256,
        /* SizeOfEventRingSeg */          256,
        /* IMod */                        0,
        /* HwResetTime */                 10,
        /* CmdAbortTime */                5000,
        /* CleanUpTime */                 3000,               4
        /* pTrbAllocPolicyFn */           NULL,
        /* ulRxThresholdRegOffset */      0,
        /* ulRxThresholdRegValue */       0,
        /* ulTxThresholdRegOffset */      0,
        /* ulTxThresholdRegValue */       0,
        /* ulSystemResumeRecoveryTime */ 10,
        /* ulMapDmaRange */               0,
        /* ulMapDmaRangeTime */           0,
        /* nGlobalBufSeg */               0,
        /* ulDebugFlags */                UDMASK_HCD |        5
                                          UDMASK_ANY |
                                          UDMASK_ERRORS
                );
```

(1) Wiring flags specify how the XHCI is wired and any silicon-specific work-arounds (in case things are not standard). See discussion below.

(2) Bus handle and I/O port are system specific. Normally bus handle is NULL.

(3) hUsbInt specifies the interrupt connection. It's specified as an Interrupt Resource Handle; these are OS and Platform Specific.

(4) These parameters configure a number of XHCI-specific parameters and express various memory space/time trade-offs. See USBPUMP_XHCD_CONFIG_INFO.

(5) These specify the kinds of DataPump logging messages to be output for this HCD instance.

About XHCI Configuration Flags

- These flags mostly enable bug workarounds or vendor-specific extensions
- For the most part, the XHCD will try to set these flags automatically based on probing the hardware
- Contact MCCI technical support if you have questions – we will supply the appropriate settings for your specific silicon

XHCI Config Flags (1)

Name (USBPUMP_XHCD_WIRING_...)	Meaning
NOT_DISABLE_IE	This flag is ignored
SET_RX_REGISTER	Set the RX threshold register. Set for TI chips, clear otherwise.
HUB_WORKAROUND_ENABLE	Enable V0.95 behavior for hosts that claim 0.96 or greater compliance
SET_ALT_PORT_POWER	For TI host controllers, set the power flags on both hub speed ports when receive a command for either hub port. *
CLEAR_P3	(Normally overridden by PCIe data) Set for TI chips, clear otherwise.
SET_TX_REGISTER	Set the TX threshold register. Set for TI chips, clear otherwise.
PORT_SPEED_WORKAROUND	Needed for TI silicon: put SS port into U0 before resetting the hardware

XHCI Config Flags (2)

Name (USBPUMP_XHCD_WIRING_...)	Meaning
DISABLE_USB2_SUSPEND	This flag is reserved
SET_MAPDMA_RANGE	Used only with ASMedia host controllers, as instructed by MCCI.
SET_ALT_PORT_WAKE_MASK	For TI host controllers: enable wake on both USB 2 and USB 3 root hub ports.
SET_P3	(Normally overridden by PCIe data) Set for TI chips, clear otherwise.
USE_PORT_MAPMPING_INFO	If not set, unconditionally use the port mapping info from configuration space (assume that it's correct)
DISABLE_WAKE_MASK	This flag is reserved
LIMIT_BINTERNAL	This flag is reserved
DISABLE_SAVE_RESTORE	This flag is reserved

XHCI Config Flags (3)

Name (USBPUMP_XHCD_WIRING_...)	Meaning
PORT_OFF_WHEN_SHUT_DOWN	Power down port when shutting down. Primarily for ASMedia XHCI.
SET_MAX_BURST	This flag is reserved
DUMMY_SHORT_TRANSFER	This flag is reserved
REPORT_COMPATIBLE_PORT	Enable dual enumeration of normal devices at both high speed and SuperSpeed. Such devices are illegal but some test fixtures are built this way.

Class Driver Initialization

- Class drivers are similarly configured, based on a header, plus a vector of individual driver descriptions.
  - Header: `USBPUMP_HOST_DRIVER_CLASS_INIT_NODE_VECTOR`
  - Description: `USBPUMP_HOST_DRIVER_CLASS_INIT_NODE`
- Each driver description includes a pointer to a “match list”, which specifies the device IDs to be supported by the driver.
  - Header: `USBPUMP_USBDI_INIT_MATCH_LIST`
  - Individual match: `USBPUMP_USBDI_INIT_MATCH_LIST_ENTRY`

USBPUMP_HOST_DRIVER_CLASS_INIT_NODE_VECTOR

```
static
CONST USBPUMP_HOST_DRIVER_CLASS_INIT_NODE_VECTOR
sk_ClassDriverInitHeader =
        USBPUMP_HOST_DRIVER_CLASS_INIT_NODE_VECTOR_INIT_V1(
                /* name of the vector */ sk_ClassDriverInitNodes,  (1)
                /* prefunction */ NULL,
                /* postfunction */ NULL                 (2)
                );
```

(1) This must be the name of an array of USBPUMP_HOST_DRIVER_CLASS_INIT_NODEs.

(2) Pre and post functions are for special purposes platform-specific initialization before and after creating the drivers. Normally not used.

USBPUMP_HOST_DRIVER_CLASS_INIT_NODE

```
static CONST USBPUMP_HOST_DRIVER_CLASS_INIT_NODE sk_ClassDriverInitNodes[] =
        {
        USBPUMP_HOST_DRIVER_CLASS_INIT_NODE_INIT_V1(
                /* pProbeFn */ NULL,                                  1
                /* pInitFn */ UsbPumpUsbdiClassMsd_Initialize,            2
6               /* pConfig */ &sk_UsbPumpUsbdiMsd_ClassConfigDefault,         3
                /* pPrivateConfig */ &gk_UsbPumpUsbdiClassMsd_ConfigDefault,      4
                /* DebugFlags */ UDMASK_ENTRY | UDMASK_ANY | UDMASK_ERRORS |
                                 UDMASK_USBDI                           5
                ),
        USBPUMP_HOST_DRIVER_CLASS_INIT_NODE_INIT_V1(
                /* pProbeFn */ NULL,
                /* pInitFn */ UsbPumpUsbdiClassComposite_Initialize,
7               /* pConfig */ &sk_UsbPumpUsbdCD_Config.ClassConfig,                8
                /* pPrivateConfig */ &sk_UsbPumpUsbdCD_Config.ClassPrivateConfig,
                /* DebugFlags */ UDMASK_ANY | UDMASK_ERRORS
                )
        };
```

(1) Probe functions allow you to disable a class driver dynamically.

(2) This is the entry point for the driver. Refer to the class driver manual.

(3) This configuration structure provides configuration info that's common for all drivers (not just Mass Storage). Provides matching, object naming, and maximum number of concurrent device instances.

(4) This optionally provides additional class-specific configuration info. Often the class driver library provides a suitable default, as in this case. Refer to class driver manual.

(5) The debug flags to be used for FUNCTION objects created by this class driver.

(6) We have two init nodes, so we're configuring 2 drivers; the first is the mass storage driver

(7) The second configured driver is the composite device driver.

(8) The composite driver uses an unusual configuration strategy, documented below; one structure provides the common and the specific info.

USBPUMP_USBDI_DRIVER_CLASS_CONFIG

```
(1) static CONST USBPUMP_USBDI_DRIVER_CLASS_CONFIG
sk_UsbPumpUsbdiMsd_ClassConfigDefault =
        USBPUMP_USBDI_DRIVER_CLASS_CONFIG_INIT_V1(
                &sk_UsbPumpUsbdiMsd_InitMatchList, (2)
                NULL, /* use implementation default driver class name */
                NULL, /* use default function instance name */
                /* max number of instances */ 1  (3)
                );
```

(1) There is one instance of this structure for each class driver, referenced from the USBPUMP_HOST_DRIVER_CLASS_INIT_NODE.

(2) The match list specifies the device IDs that will be matched by this driver.

(3) This specifies how many instances are concurrently supported. Each instance consumes memory. (Note: it is possible to configure the USBD to dynamically allocate function instances when devices arrive, and to free them when devices depart. However, this means that the USBD operation is not deterministic – device enumeration may fail if the system is low on memory.

Device IDs

- Device/driver matches are specified using strings.
- The hub driver prepares a string that represents the device; the string is of the following form
  - **`vid=XXXX/XXXX;r=XXXX;[dc=xx/yy/zz;][if=xx;][fc=xx/yy/zz;][ig=gg;][mf;]`**
- All numbers are formatted in lower-case hexadecimal.
  - "vid=XXXX/XXXX" is the vendor ID/product ID
  - "r=XXXX "is the revision ID
  - "dc=xx/yy/zz" is the device class/subclass/protocol. This is omitted if the class/subclass/protocol are all zero, unless the function class/subclass/protocol are also all zero.
  - "if=xx" is the interface number (in hex, two digits)
  - "fc=xx/yy/zz" is the function class/subclass/protocol. This may be taken from the interface descriptor or from the IAD.
  - "ig={xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}" is the interface GUID, and is only present for functions whose primary interface is an MDLM interface. The GUID is formatted in normal form, i.e. with '{', '}' and embedded '-'
  - "mf" is the multi-function device indicator. It's only present on devices enumerated by the composite driver.
- Special cases:
  - The device class/subclass/protocol will be omitted if they are "dc=00/00/00".
  - The interface number "if=xx" will be present only for true multi-function devices. The function class/subclass/protocol "fc=xx/yy/zz" will always be present.

Device ID Matching

- Each match-list entry contains a matching pattern
- This pattern is matched using UsbPumpLib_MatchPattern()
  - ‘*’ is a wildcard pattern
  - The matches are “anchored” – the pattern must match the entire string
- Some examples:
  - "**`vid=040e/f101;*;if=03;*`**"

    matches interface 3 on device with vendor ID/product ID 040e/f101. The device revision and class, and the function class are ignored.
  - "**`*;fc=08/06/50;*`**"

    Matches any function of class 8, subclass 6, protocol 0x50. This is the matching string for mass storage BOT encapsulated SCSI.

Match priorities

- Like most USB stacks, TrueTask USB uses a priority system to resolve situations where a device is matched by multiple drivers.
- All drivers are checked, and the match with the highest priority is used
- Standard priority values are defined by MCCI, but you can use your own if you like (there are intentionally gaps in the sequence)
- Note that TrueTask USB **does not** parse the match patterns; it's the programmer's responsibility to write meaningful patterns and use meaningful match priorities
- Match priorities are enumerated in the following two slides.

Match priority values

USBPUMP_USBDI_-PRIORITY_…	Value	Conventional use
VIDPIDREV	0xF000	Highest priority match: used for vid/pid/rev matches
VIDPID	0xE000	Vid/pid matches
VIDPIDREV_IF	0xD800	Vid/pid/if=nn matches
VIDPID_IF	0xD000	If=nn matches
DEV_CSP	0xC000	Device class/subclass/protocol matches
DEV_CS	0xB000	Device class/subclass matches
DEV_C	0xA000	Device class matches
VIDPIDREV_FN_CSP	0x9800	Vid/pid/rev and function class/subclass/protocol matches
VIDPID_FN_CSP	0x9000	Vid/Pid and function class/subclass/protocol
VIDPIDREV_FN_CS	0x8800	Vid/Pid/Rev and function class/subclass matches

Match priority values

USBPUMP_USBDI_-PRIORITY_...	Value	Conventional use
VIDPID_FN_CS	0x8000	Vid/Pid and Function class/subclass matches
VIDPIDREV_FN_C	0x7800	Vid/Pid/Rev and Function class matches
VIDPID_FN_C	0x7000	Vid/Pid and Function class matches
FN_GUID	0x6000	Function GUID matches
FN_CSP	0x5000	Function class/subclass/protocol matches
FN_CS	0x4000	Function class/subclass matches
FN_C	0x3000	Function class matches
WEAK	0x1000	Weak – no specific priority for the match; used only if no more specific match triggers

USBPUMP_USBDI_INIT_MATCH_LIST

```
(1) static CONST USBPUMP_USBDI_INIT_MATCH_LIST
    sk_UsbPumpSampleMsc_InitMatchList =
            USBPUMP_USBDI_INIT_MATCH_LIST_INIT_V1(
                    sk_vUsbPumpSampleMsc_Matches  (2)
                    );
```

(1) There is one instance of this structure for each class driver, referenced from the USBPUMP_USBDI_DRIVER_CLASS_CONFIG structure.

(2) The macro requires the name of an array of USBPUMP_USBDI_INIT_-MATCH_LIST_ENTRY elements. This must be in the local file, as the macro uses the size of the array to initialize the USBPUMP_USBDI_INIT_MATCH_-LIST.

USBPUMP_USBDI_INIT_MATCH_LIST_ENTRY

```
static CONST USBPUMP_USBDI_INIT_MATCH_LIST_ENTRY
sk_vUsbPumpSampleMsc_Matches[] =
        {
        USBPUMP_USBDI_INIT_MATCH_LIST_ENTRY_INIT_V1(    ①
                "*;fc=08/06/50;*", /* msc,scsi,BOT */
                USBPUMP_USBDI_PRIORITY_FN_CSP ②
                ),
        USBPUMP_USBDI_INIT_MATCH_LIST_ENTRY_INIT_V1(
                "*;fc=08/02/50;*", /* msc,atapi,BOT */   ③
                USBPUMP_USBDI_PRIORITY_FN_CSP
                ),
        USBPUMP_USBDI_INIT_MATCH_LIST_ENTRY_INIT_V1(
                "*;fc=08/05/50;*", /* msc,SFF-8070i,BOT */④
                USBPUMP_USBDI_PRIORITY_FN_CSP
                )
        };
```

(1) There is one instance of this structure for each class driver, referenced from the USBPUMP_USBDI_DRIVER_CLASS_CONFIG structure.

(2) The first entry matches any function marked as BOT with SCSI commands.

(3) The priority of this (and all matches in this list) is “class/subclass/protocol at the function level”.

(4) The second entry matches any function marked as BOT with ATAPI commands.

(5) The third entry matches any entry marked as BOT with SFF-8070i commands.

Configuring the Composite Driver

- To enable composite device support, you must include the composite driver in the set of known drivers.
- To conserve RAM, you inform the composite driver about the intended complexity of the devices you wish to support.
- As with the Windows stack, the composite driver will create a virtual USB device for each detected composite function.

USBPUMP_USBDI_CLASS_COMPOSITE_CONFIG

```
static
CONST USBPUMP_USBDI_CLASS_COMPOSITE_CONFIG sk_UsbPumpUsbdCD_Config =
  USBPUMP_USBDI_CLASS_COMPOSITE_CONFIG_INIT_V1(
    &gk_UsbPumpUsbdiClassComposite_InitMatchList, ①
    /* driver class name */ NULL, /* use implementation default */
    /* function instance name */ NULL, /* use default */
    /* number of instances */ 8,  ②
    /* number of Composite Functions Total */ 64, ③
    /* number of Composite Sub Function per Function */ 32, ④
    /* number of Pipes */ 12, ⑤
    /* size of Configuration Desc */ 512  ⑥
    );
```

(1) TrueTask USB includes the appropriate init match list for composite devices.

(2) We allow 8 composite devices concurrently – obviously, this is for a big system.

(3) We allow a total of 64 composite functions spread across those 8 devices

(4) We allow 32 interfaces per composite function

(5) We allow 12 pipes per composite device

(6) We allow 512 bytes for the config descriptor buffer used by the composite device.

DATAPUMP LIBRARY FACILITIES

Library Facilities

- Formatting Strings
- Descriptor Support
- UNICODE® Support (UTF-8, UTF-16, UTF-32)

Terry Moore:

In addition, we need info about circular queue (usbpumplist.h functions), string matching fns, UHIL_{LE,BE}_{GET,PUT}UINT*

Formatting data into strings

- A common operation, but often either not supplied by compiler vendors in a useful form for embedded work
- MCCI has a fast, reentrant portable printf()-like routine with extensions for common USB operations
- All the normal printf-like APIs are available except floating point.
- Also can be used with callbacks to capture formatted output

Formatting Functions

- "usbdebug.h" is the common header file for the text formatting functions.
  - Name is misleading: these routines do not cause I/O to happen! Use "usbpumpdebug.h" (and TTUSB_PRINTF(), TTUSB_PLATFORM_PRINTF(), etc.)
- UsbDebugVprintf() is the low-level primitive; it calls a user-supplied function to process formatted output.
- UsbDebugSnprintf(), UsbDebugVsnprintf() format into a user-supplied buffer

Prototypes

- int UsbDebugSnprintf(
```
        TEXT *pOutBuf,
        BYTES size_OutBuf,
        CONST TEXT *pFormat,
        …
        );
```
- typedef BYTES TTUSB_VNPRINTF_FN(
```
                        VOID *pCtx,
                        CONST TEXT *pBuf,
                        BYTES nBuf
                        );
```
- int UsbDebugVprintf(
```
        TTUSB_VNPRINTF_FN *pOutputFn,
        VOID *pCtx,
        CONST TEXT *pFormat,
        va_list arg_pointer
        );
```

Format Specifiers

- All the non-floating point specifiers from ANSI C (1987)
- The non-floating point width specifiers from ISO C (1999):
  - %zu – size_t width unsigned
  - %llu – long long width %u
  - %jx – intmax_t width %x
  - %tx – ptrdiff_t width %x
  - %hhu – “short short” (char-width, in practice) %u
- Additional MCCI extensions:
  - “%S” (capital S) for formatting USB (little-endian) UNICODE into ASCII.
  - “%b” for specifying strings by base and length. Two arguments; first is pointer, second is byte count. Can display embedded NULLs
  - “%B” for formatting USB (little-endian) UNICODE into ASCII, from counted buffer. Two args, first is pointer, second is character count (2*byte count).

Harvard Architecture Variants

- On Harvard architecture CPUs, strings can be put in "ROM" space to save room.
  - But you need special pointer types
- The primitives UsbDebugSnprintf_p() and UsbDebugVprintf_p() are identical to UsbDebugSnprintf() and UsbDebugVprintf(), but the format string is in code space

USB Descriptor Handling

- Structures are defined in “usbdesc.h”
- Two forms:
  - Wire format – bytes, as transmitted
  - Internal formats – assumes that someone has translated multibyte fields into native USHORTs, ULONGs, etc.
- Be careful about packing / alignment issues
  - Use the **`LENGTHOF_…`** macros to find the size of the wire descriptors. Example: use **`LENGTHOF_USBIF_DEVDESC_WIRE`** rather than **`sizeof(USBIF_DEVDESC_WIRE)`**

Parsing Descriptors

- For class drivers, always needed
- For device implementations, sometimes necessary because of limitations of USB resource compiler – best way to ensure coherence in setups is to parse the descriptors in the device.
- A comprehensive library is provided:
  - UsbFindNextDescriptorInConfig()
  - UsbFindNextDescriptorInInterface()
  - UsbParseConfigurationDescriptor() – flexible search capabilities