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

MCCI USB DataPump Embedded Host and OTG User's Guide

Engineering Report 950000327
Rev. E
Date: 2011/09/28

PROPRIETARY NOTICE AND DISCLAIMER

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

Document Release History

Rev. A	2008/10/01	Original release
Rev. B	2008/10/02	Revision number
Rev. C	2010/02/23	Fixed technical problem in pdf
Rev. D	2010/08/16	Changed all references to Moore Computer Consultants, Inc. to MCCI Corporation. Changed document numbers to nine digit versions.
Rev. E	2011/09/28	Added source code disclaimer.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 Introduction

This manual introduces the “MCCI USB DataPump Embedded Host” and “MCCI USB DataPump On-The-Go” products, referred to as “Embedded Host/OTG” in this document, for simplicity. The main intent of this manual is to provide an overview of the Embedded Host/OTG and, more specifically, provide guidance on where additional information and configuration details can be found in lower level documents.

MCCI’s Embedded Host/OTG is a fully upwards-compatible enhancement to the current MCCI USB DataPump, for adding host support to embedded products. OTG is a supplement to the USB 2.0 specification that allows USB peripherals to communicate directly with selected other USB peripherals through limited host capability.

1.1 Glossary

Brand	MCCI’s term for the concrete set of drivers derived from the MCCI core library with changes as specified by the customer
class driver	An instance of "driver class" for a particular "device class”.
class driver instance	An instance of a “class driver”.
Composite device	A specific way of representing a USB device that supports multiple independent functions concurrently. In this model, each USB Function consists of one or more interfaces, with the associated endpoints and descriptors. In the DataPump environment, the parent driver divides the composite device up into single functions, and then uses standard object-oriented techniques to present the descriptors of each function to the function drivers. This allows function drivers to be coded the same way whether they are running as the sole function on a device or as part of a multi-function composite device. Compare with “compound device” as defined in [USBCORE].
DCD	*See* Device Controller Driver
device class	Class a USB device is categorized into, such as hub, human interface, printer, imaging, or mass storage device.
Device controller	The hardware module responsible for connecting a USB device to the USB bus.
Device Controller Driver (DCD)	The software component that provides low-level access to the specific Device Controller in use. All MCCI USB DataPump DCDs implement a common API, allowing the rest of the DataPump device stack to be hardware independent.

device stack	Collective term for the software stack that implements USB device functionality.
driver class	A generic representation of a USB device driver.
EH	Embedded Host
HCD	*See* Host Controller Driver
HCD Class	*See* Host Controller Driver
HCD Instance	*See* Host Controller Driver
Host controller	The hardware module responsible for operating the USB bus as a host.
Host Controller Driver	The software component that provides low–level access to the specific Host Controller in use. This term may refer a specific instance of the software that models the host controller to upper layers of software, or it may refer to the entire collection of code that implements the driver. Where necessary, we refer to the collection of code as the "HCD Class", and the specific data structures and methods that represent a given instance as an "HCD Instance".
host stack	Collective term for the software stack that implements USB host functionality.
OTG	Abbreviation for USB On-The-Go
OTGCD	*See* OTG Controller Driver
OTG Controller	*The* hardware module responsible for operating a dual-role OTG connection.
OTG Controller Driver	The software component that provides low-level access to a USB bus via an OTG Controller. Normally export three APIs, an HCD API, a DCD API, and a (shared) OTG
OTG stack	device stack + host stack
Phy	Short for "physical layer". Often used as short-hand for "transceiver". MCCI uses this in the abbreviations for the API operations that are used for accessing the phy.
Transaction Translator	A functional component of a USB hub. The Transaction Translator responds to special high-speed transactions and translates them to full/low-speed transactions with full/low-speed devices attached down downstream facing ports.
Transceiver	The hardware module responsible for low-level signaling on the USB bus.
USBD	USB Driver, the generic term for the USB Management module.

USBDI — USB Driver Interface, the generic term for the API between USB function drivers and USBD.

xCD — Host, Device, Dual-Role or OTG Controller Driver

1.2 Referenced Documents

[DPUG] *MCCI USB DataPump Users Guid*e, MCCI Engineering report 950000066

[USBCORE] *Universal Serial Bus Specification*, version 2.0 / 3.0 (also referred to as the USB Specification), with published erratas and ECOs. This specification is available on the World Wide Web site http://www.usb.org/.

1.3 Target Audience

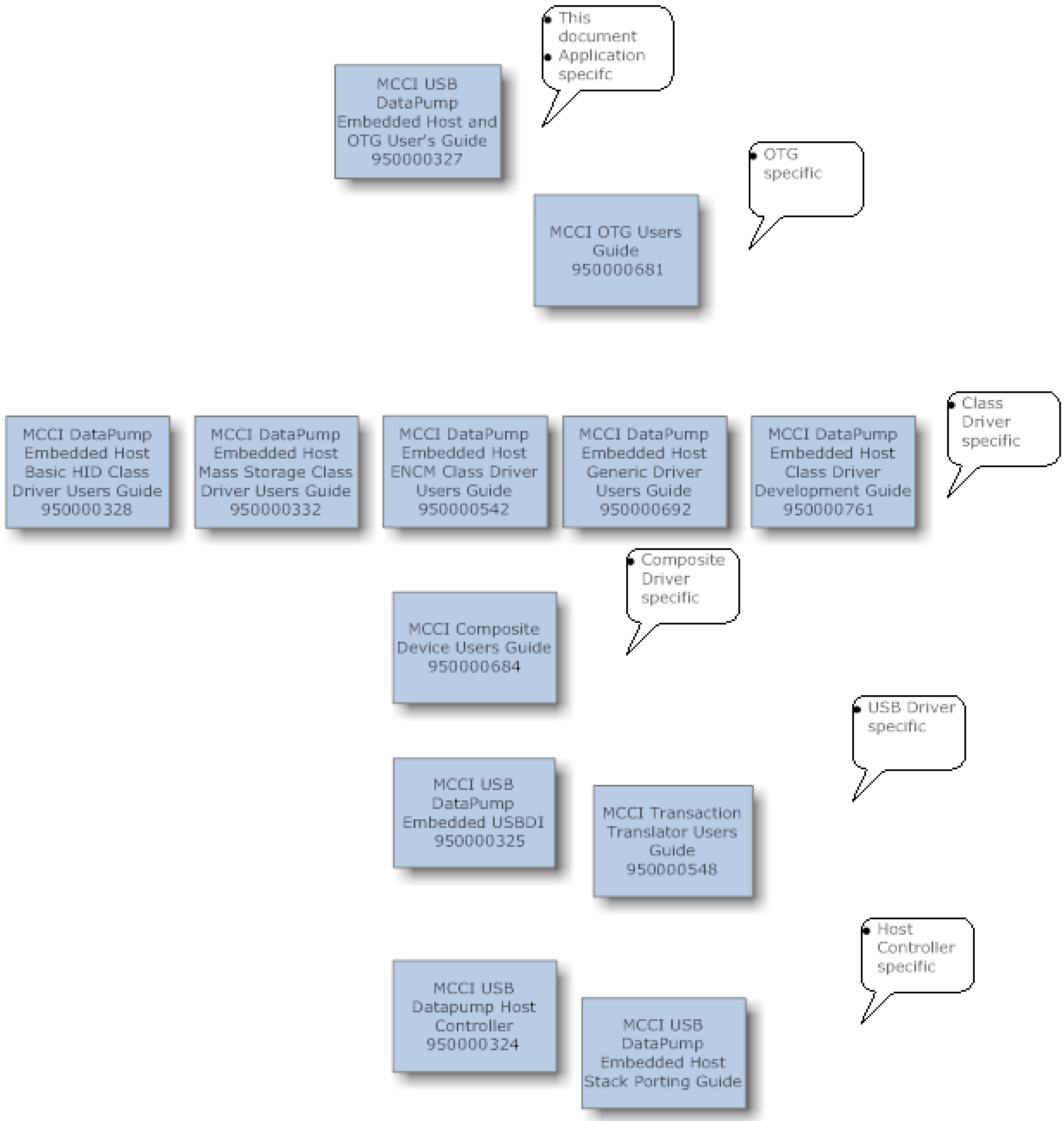
The *MCCI USB DataPump Embedded Host and OTG User's Guide* is for customers who intend to develop application, class driver, and/or hardware interface software to create a USB embedded host or OTG system.

This document assumes familiarity with the MCCI USB DataPump.

1.4 Documentation Tree

This manual, the *MCCI USB DataPump Embedded Host and OTG User's Guide,* is the primary manual in the set of manuals for the MCCI USB Embedded Host/OTG Firmware Development Kit. Figure 1-1 graphically shows the list of all MCCI created manuals related to the MCCI USB Embedded Host/OTG.

Figure 1-1 MCCI USB Embedded Host /OTG Documentation Tree



1.5 Related Documentation

The following is a list of documentation installed automatically as part of the distribution. The documentation is contained in the MCCI/tools/doc directory. It can also be easily referenced using any web browser by accessing the MCCI customer secure site. You must have Adobe Acrobat Reader installed to open many of the files there.

Table 1 MCCI USB Embedded Host/OTG documents

Documentation Subject	Document Number
MCCI USB DataPump Embedded Host and OTG User's Guide (this document)	950000327
MCCI USB DataPump Host Controller	950000324
MCCI USB DataPump Embedded Host Stack Porting Guide	
MCCI USB DataPump Embedded USBDI	950000325
MCCI Transaction Translator Users Guide	950000548
MCCI Composite Device Users Guide	950000684
MCCI DataPump Embedded Host Class Driver Development Guide	950000761
MCCI DataPump Embedded Host Basic HID Class Driver Users Guide	950000328
MCCI DataPump Embedded Host Mass Storage Class Driver Users Guide	950000332
MCCI DataPump Embedded Host ENCM Class Driver Users Guide	950000542
MCCI DataPump Embedded Host Generic Driver Users Guide	950000692
MCCI OTG Users Guide	950000681
MCCI USB DataPump User's Guide	950000066

2 Product Overview

The following product variants are offered:

- Full OTG package with host and device support
- Host-only package
- OTG-enabled host stack for integration with customer's device stack (which needs to be evaluated for integration suitability)
- OTG-enabled host upgrade for older MCCI device stack customers

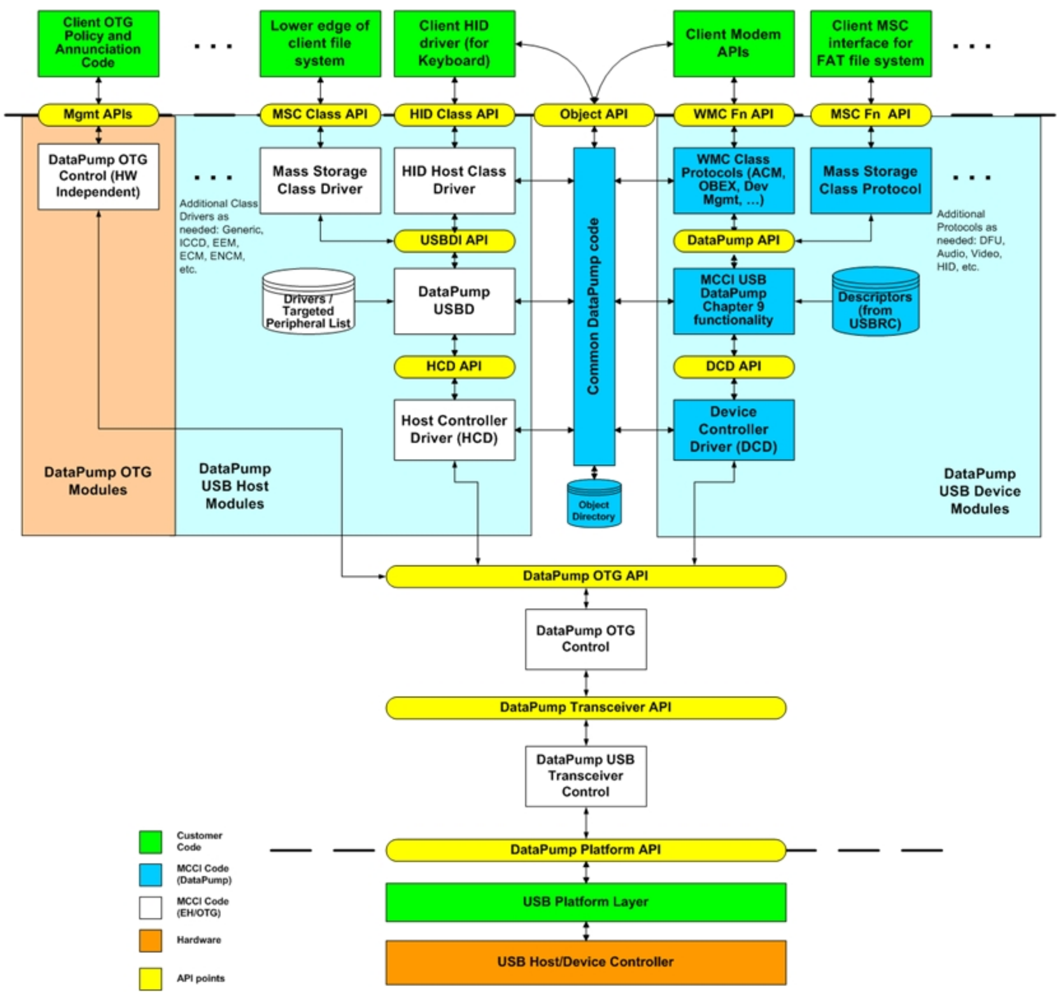
MCCI does not offer OTG-only support for integration with 3rd party host and device stacks.

Extensive device class support is available, including:

- Hub (with or without Transaction Translator support)
- Composite Device
- Generic (for working with external applications)
- Mass Storage (MSC)
- Human Interface (HID)
- Networking (ECM, ENCM)
- ACM/WMC (planned)
- USBSIM Classes (UICC, EEM) (planned)

3 Embedded Host/OTG Firmware Architecture

Figure 3-1 Embedded Host/OTG Firmware Architecture



The MCCI Embedded Host/OTG stack is a fully upwards-compatible enhancement of the current MCCI USB DataPump product. As such, all current USB device functionality is preserved, not only in terms of the code written by MCCI for device class support, but also in terms of the code written by clients for the existing platforms.

The Embedded Host/OTG stack consists of the base MCCI USB DataPump, plus the MCCI USB DataPump Embedded Host (EH) Modules, plus the USB On-The-Go specific functionality needed for handling the switch between host and device mode. This breakdown is shown in Figure 3-1.

The Embedded Host/OTG stack is implemented as an event-driven, message based API for use by class drivers. Class Drivers may be written in ways similar to the DataPump Device Class Protocols (in which case they will run on an event driven basis inside the DataPump OTG thread) or else they may be written as separate threads, in which case they will communicate with a generic stub driver that runs inside the DataPump. Driver selection is controlled by a database, called the “Driver Directory / Targeted Device List”. Drivers can match on a class or vendor ID/product ID basis. Driver matching is separate from driver code.

To preserve RAM, Hub support is preconfigured in terms of the number of hubs and number of ports that can be supported at one time. Similarly, each supported class will need to be preconfigured by the platform engineer to select the number of instances that will be supported. Hub support is not shown in Figure 3-1, but it is implemented as part of the USBD. There are no external APIs for hubs.

The Composite Device Driver (not shown in Figure 3-1) is layered between the USBD and the class drivers. An instance is automatically invoked when a multi-function or multi-configuration device is plugged in; it is not invoked for a single-function device. The Composite Device Driver is invisible to the class driver, because the API for the composite driver is the same as the API for the USBD. The class driver has no knowledge of how many other functions co-exist in the same device or whether the Composite Device Driver layer is there.

Transaction Translator support, if enabled, is implemented as part of the USBD.

The API points for the Embedded Host/OTG stack, shown in Figure 3-1, are described in separate manuals. See Table 2 for reference.

Table 2. API/Manual reference

API point	Manual(s) the APIs are documented in	Document Number
Mgmt APIs DataPump OTG API	MCCI OTG Users Guide	950000681
MSC Class API	MCCI DataPump Embedded Host Mass Storage Class Driver Users Guide	950000332
HID Class API	MCCI DataPump Embedded Host Basic HID Class Driver Users Guide	950000328
ENCM Class API	MCCI DataPump Embedded Host ENCM Class Driver Users Guide	950000542

API point	Manual(s) the APIs are documented in	Document Number
(not shown in figure)		
Generic Class API (not shown in figure)	MCCI DataPump Embedded Host Generic Driver Users Guide	950000692
Composite Device Driver API (not shown in figure)	MCCI Composite Device Users Guide	950000327
USBDI API	MCCI USB DataPump Embedded USBDI MCCI Transaction Translator Users Guide	950000325 950000548
HCD API	MCCI USB DataPump Host Controller	950000324

4 Implementing a Custom Embedded Host/OTGApplication

4.1 Overview

To write a custom Host/OTG application, you should start by looking at the sample application code that is included in your kit. See usbkern/host/app/otghidmscapp and hosthidmscapp. The sample OTG application is otghidmscapp.exe. The sample Host application is hosthidmscapp.exe. Both sample applications, run in the Windows environment with the catena1650 card. Both sample applications are hardware dependant.

You will need to write similar software for the platform and USB chip you want to support.

The following information is required in order to configure your application:

1. What platform will I use?
2. What USB chip will I use?
3. Do I need Transaction Translator support?
4. Do I need Composite Driver Support?
5. What Class Drivers do I want to include?
6. Do I want to enable client notification?

Note that section 4.4 through the end of this document, apply to both OTG and Host applications. If you want to create an OTG application, you should read section 4.2 and then skip to section 4.4 and read through the rest of the document. If you want to create a Host application, you should read section 4.3 through to the end of the document.

4.2 OTG Application Specific Initialization

The initialization of the DataPump is table driven. At system startup time, a routine named `UsbPump_GenericApplicationInit` runs through a tree of device structures and initializes and starts the device-side of the DataPump. The description of how to fill in the tree of device structures is in document *950000066-MCCI USB DataPump User's Guide*. The sample root device structure is named `gk_UsbPumpApplicationInitHdr` and is located in usbkern\host\app\otghidmscapp\otghidmscapp_device_appinit.c.

```
CONST USB_DATAPUMP_APPLICATION_INIT_VECTOR_HDR gk_UsbPumpApplicationInitHdr =
        USB_DATAPUMP_APPLICATION_INIT_VECTOR_HDR_INIT_V1(
            sk_UsbPumpApplicationInitVector,
            /* pSetup */     OtgHidMscApp_Device_Setup,
            /* pFinish */    OtgHidMscApp_Device_InitFinish
            );
```

Instructions on how to fill in `sk_UsbPumpApplicationInitVector` can be found in document *950000066-MCCI USB DataPump User's Guide.* In order to enable host support, you need to do the following things:

1. Call the PHY initialization function, `UsbPump_GenericPhyInit` from your device pre-processing function `(OtgHidMscApp_Device_Setup)`.
2. Call the host initialization function, `UsbPump_GenericHostInit`, at the end of your device post-processing function `(OtgHidMscApp_Device_InitFinish)`.

The `UsbPump_GenericPhyInit` function runs through a tree of PHY structures to initialize the PHY (USB transceiver). All USB devices and hosts are (at least logically) connected to the bus port using USB transceivers. However, only host-only and OTG applications are required to create a PHY tree of structure. The function, `UsbPump_GenericPhyInit`, is located in usbkern\phy\common\usbphy_genericinit.c. It is strategically located in a common directory so that device-only implementations could potentially use it in the future.

The `UsbPump_GenericHostInit` function runs through a tree of host structures and initializes and starts the embedded host system.

Instructions on how to fill in the PHY tree of structures and the host tree of structures are in the subsequent sections. (Note that names beginning with "gk" indicate global konstant (as opposed to char) and names beginning with "sk" indicate static konstant.)

4.3 Host Application Specific Initialization

The initialization of the DataPump is table driven. At system startup time, a routine named `UsbPump_GenericApplicationInit` runs through a tree of device structures and initializes and starts the device-side of the DataPump. Since, in this case, the device-side of the DataPump is not desired, the device-side initialization must be stubbed out.

The sample root device structure is named `gk_UsbPumpApplicationInitHdr` and is located in usbkern\host\app\hosthidmscapp\hosthidmscapp_appinit.c.

```
CONST USB_DATAPUMP_APPLICATION_INIT_VECTOR_HDR gk_UsbPumpApplicationInitHdr =
          USB_DATAPUMP_APPLICATION_INIT_VECTOR_HDR_INIT_V1(
              sk_UsbPumpApplicationInitVector,
              /* pSetup */     NULL,
              /* pFinish */    HostHidMscApp_InitFinish
```

Note the application initialization vector is named `sk_UsbPumpApplicationInitVector`. This is where the stubbing out of the device-side takes place. If `sk_UsbPumpApplicationInitVector` is initialized as described below, the device-side will not be enabled.

```
static
CONST USB_DATAPUMP_APPLICATION_INIT_VECTOR sk_UsbPumpApplicationInitVector[] =
          {
          USB_DATAPUMP_APPLICATION_INIT_VECTOR_INIT_V1(\
              /* port index */ -1,             \
              /* pDescriptorTable */ NULL,     \
              /* pDeviceInitFunction */ NULL,  \
              /* sizeof_Udevice */ 0,          \
              /* DebugFlags */ 0,              \
              /* pAppProbeFunction */ NULL,    \
              /* pAppInitFunction */ NULL,     \
              /* pAppInfo */ NULL              \
              )
          };
```

From this point on, initialization of a host-only application is the same as the initialization of an OTG application.

In order to enable host support, you need to do the following things:

1. Call the PHY initialization function, `UsbPump_GenericPhyInit` from your device pre-processing function `(HostHidMscApp_InitFinish)`.
2. Call the host initialization function, `UsbPump_GenericHostInit`, at the end of your device post-processing function `(HostHidMscApp_InitFinish)`.

The `UsbPump_GenericPhyInit` function runs through a tree of PHY structures to initialize the PHY (USB transceiver). All USB devices and hosts are (at least logically) connected to the bus port using USB transceivers. However, only host-only and OTG applications are required to create a PHY tree of structure. The function, `UsbPump_GenericPhyInit`, is located in usbkern\phy\common\usbphy_genericinit.c. It is strategically located in a common directory so that device-only implementations could potentially use it in the future.

The `UsbPump_GenericHostInit` function runs through a tree of host structures and initializes and starts the embedded host system.

Instructions on how to fill in the PHY tree of structures and the host tree of structures are in the subsequent sections. (Note that names beginning with “gk” indicate global konstant (as opposed to char) and names beginning with “sk” indicate static konstant.)

4.4 Filling in the PHY Tree of Structures

The root PHY structure is `gk_UsbPumpPhy_GenericInitVector`, and is located in arch/i386/port/catena1650/common/cat1650_usbphy.c. Note that this structure is required for both OTG and host-only applications.

```
CONST USBPUMP_USBPHY_INIT_NODE_VECTOR gk_UsbPumpPhy_GenericInitVector =
        USBPUMP_USBPHY_INIT_NODE_VECTOR_INIT_V1(
                /* name of the vector */ PhyInitNodes,
                /* prefunction */ NULL,
                /* postfunction */ NULL
                );
```

The root PHY structure contains a vector of PHY initialization nodes, as well as pre-processing and post-processing functions. The pre-processing function runs before the PHYs are initialized and the post-processing function runs after the PHYs are initialized.

Based on the platform/PHY(s) you plan to use, the appropriate `gk_UsbPumpPhy_GenericInitVector` should be used.

Below is the definition for the PHY used in the sample application. In this case, only one PHY is defined, but you have the option of defining multiple PHYs.

```
static
CONST USBPUMP_USBPHY_INIT_NODE PhyInitNodes[] =
        {
        USBPUMP_USBPHY_INIT_NODE_INIT_V1(
                /* pProbeFn */ NULL,
                /* pInitFn */ Isp1761UsbPhy_CreateV2,
                /* pConfig */ &gk_Isp1761UsbPhy_ConfigInfo,
                /* pPrivateConfig */ &gk_Cat1650_UsbPhyISP1761ConfigInfo,
                /* DebugFlags */ UDMASK_ANY | UDMASK_ERRORS | UDMASK_HWEVENT
                )
        };
```

This structure contains a pointer to the PHY initialization function, a pointer to the PHY configuration, a pointer to the private PHY configuration, debug flags and a pointer to the probe function (not used in the sample application), to further qualify which PHY to use.

The PHY configuration `(USBPUMP_USBPHY_CONFIG_INFO *pConfig)` defines the configuration for the PHY itself. It is defined in the chip driver that is common to all platforms.

The private PHY configuration defines the connectivity between the platform and the PHY and is defined in the platform layer in the customer environment.

The sample applications use the Catena1650 platform and the ISP1761 PHY. Depending on what platform/PHY you use, you will need to set the pointers to the appropriate PHY configuration structures.

The PHY initialization function, pointed to by `pInitFn`, takes the PHY configuration and private PHY configuration as input. The sample application PHY initialization function is `Isp1761UsbPhy_CreateV2`. See isp1761_create2.c and isp1761_create.c, located in usbkern\ifc\isp1761\phy. By looking at the source code, you can see that the call stack is:

```
UsbPumpUsbPhy_Create
Isp1761UsbPhy_Create↑
Isp1761UsbPhy_CreateV2↑
```

`UsbPumpUsbPhy_Create` is the generic function to create a PHY object instance. (See usbkern\phy\common\usbphy_create.c.) It takes a pointer the PHY configuration as an argument. In addition to creating the PHY object instance, it initializes the OTG finite-state-machine (FSM) using `UsbPumpOtgFsm_Initialize_V2`. (See usbkern\common\otgfsm.c.)

The OTG FSM initialization function will create the OTGFSM annunciator object with the specified number of sessions and it will open the annunciator sender session. In the case of a host-only application, the OTG FSM will not progress.

The following sections give instructions on how to fill in the PHY configuration structures.

4.4.1 Filling in the PHY Configuration Structure

Below is the PHY configuration for the sample applications. See usbkern\ifc\isp1761\phy\isp1761_create.c. The PHY configuration is chip driver specific, but is common for all platforms. Currently, only the MCCI chip porting engineer is responsible for filling this in.

```
CONST USBPUMP_USBPHY_CONFIG_INFO  gk_Isp1761UsbPhy_ConfigInfo =
          USBPUMP_USBPHY_CONFIG_INFO_INIT_V2(
              /* UsbPhyObjectSize */ sizeof(USBPUMP_USBPHY_ISP1761),
              /* UsbPhyObjectName */ USBPUMP_USBPHY_ISP1761_NAME,
              /* NumPorts */         USBPHY_ISP1761_NUM_PORTS,
              /* tb_vbuschrg_srp */  USBPHY_ISP1761_TB_VBUSCHRG_SRP,
              /* pUsbPhyCallbackFn */Isp1761UsbPhy_IoctlQeHandler,
              /* pOtgFsmCallbackFn */Isp1761UsbPhy_OtgFsmEventHandler,
              /* pObjectIoctlFn */   Isp1761UsbPhy_Ioctl,
              /* AnnunciatorMaxSession */ 4
              );
```

Table 3. PHY Configuration Contents

Field Name	Description
`UsbPhyObjectSize`	Size of the PHY object that is to be created
`UsbPhyObjectName`	Name of the PHY object that is to be created
`NumPorts`	Number of ports on the PHY
`tb_vbuschrg_srp`	Timeout value for VBUS discharging in millisecond. Used by the OTG FSM. This value depends on the power supply and the capacitance. It can not be more than 100 because of the TB_SRP_INIT contraint, and normally

Field Name	Description
	must be much less. If your system can hold millisecond timer accuracy, 30ms should work in every case. But if not, you'll have to adjust the value used here.
`pUsbPhyCallbackFn`	Pointer to IOCTL Queue handler function.Used for asynchonous IOCTL processing.
`pOtgFsmCallbackFn`	Pointer to OTG FSM event handler function for this PHY.
`pObjectIoctlFn`	Pointer to PHY IOCTL handling function.
`AnnunciatorMaxSession`	Number of OTG Annunciator sessions to create.

IOCTLs targeted at the PHY object, are handled by the `UsbPumpUsbPhy_Ioctl` generic function, located in usbkern/phy/common/usbphy_ioctl.c. This function first calls the PHY specific IOCTL handler for all USBPUMP_USBPHY_ISP1761 instances, `Isp1761UsbPhy_Ioctl` in this case, to process PHY specific IOCTLs. If the IOCTL is not claimed, `UsbPumpUsbPhy_Ioctl` will check to see if there is generic processing for the PHY IOCTL. If not, the IOCTL is passed to the OTGFSM IOCTL handler.

4.4.2 Filling in the Private PHY Configuration Structure

Below is the private PHY configuration for the sample applications. See usbkern\arch\i386\port\catena1650\common\cat1650_usbphy.c. The private PHY configuration structure defines the connectivity between the platform and the PHY. You will need to create a structure like this in your platform environment (e.g. usbkern\arch*\port*\common*_usbphy.c).

```
CONST USBPUMP_USBPHY_ISP1761_CONFIG_INFO   gk_Cat1650_UsbPhyISP1761ConfigInfo =
          USBPUMP_USBPHY_ISP1761_CONFIG_INFO_INIT_V4(
              /* ulWiring */0,
              /* hISP1761Int */ (UHIL_INTERRUPT_RESOURCE_HANDLE) &gk_Cat1650_IntInfo,
              /* pPrimaryISP1761Isr */ Cat1650_UsbPhyPrimaryIsr,
              /* DebugFlags */ UDMASK_ERRORS | 0,
              /* PortMaxPower */ 500,
              /* hBus */ (UHIL_BUSHANDLE) &gk_Cat1650_BusInfo,
              /* IoPort */ (IOPORT) 0,
              /* ulHwModeCtl */ ISP1761_HWMODCTL_ADOC,
              /* pSetUsbInterruptFn */ Cat1650Phy_SetUsbInterrupt,
              /* pGetOtgModeFn */ Cat1650Phy_GetOtgMode,
              /* pInitHardwareFn */ Cat1650Phy_InitHardware,
              /* pGetDeviceInfoFn */ Cat1650Phy_GetDeviceInfo
              );
```

Table 4. Private PHY Configuration Contents

Field Name	Description
`ulWiring`	Flags used to communicate hardware design decisions to the chip driver. Not used in USB host system. Set to 0.
`hISP1761Int`	Interrupt Resource Handle
`pPrimaryISP1761Isr`	Primary PHY ISR

Field Name	Description
DebugFlags	Debug flags
PortMaxPower	Maximum power
hBus	Bus Handle
IoPort	IO Port
ulHwModeCtl	Hardware Mode Control
pSetUsbInterruptFn	Re-enable ISP1761 interrupt
pGetOtgModeFn	Get OTG mode
pInitHardwareFn	Initialize hardware
pGetDeviceInfoFn	Get Device Information

4.5 Filling in the Host Tree of Structures

The root host structure is `gk_UsbPumpHost_GenericInitVector`, of type `USBPUMP_HOST_INIT_NODE_VECTOR`.

See usbkern\host\app\otghidmscapp\otghidmscapp_host_appinit.c
(or hosthidmscapp\hosthidmscapp_appinit.c).

Below is the initialization of this structure for the otghidmscapp.exe sample application.

```
CONST USBPUMP_HOST_INIT_NODE_VECTOR gk_UsbPumpHost_GenericInitVector =
          USBPUMP_HOST_INIT_NODE_VECTOR_INIT_V1(
              /* name of the host init vector */ sk_HostInitNodes,
              /* prefunction */ NULL,
              /* postfunction */ OtgHidMscApp_Host_InitFinish
              );
```

Below is the initialization of this structure for the hosthidmscapp.exe sample application, which is identical to the one for the otghidmscapp.exe sample application, except for the name of the post-processing function.

```
CONST USBPUMP_HOST_INIT_NODE_VECTOR gk_UsbPumpHost_GenericInitVector =
          USBPUMP_HOST_INIT_NODE_VECTOR_INIT_V1(
              /* name of the vector */ sk_HostInitNodes,
              /* prefunction */ NULL,
              /* postfunction */ HostHidMscApp_Host_InitFinish
              );
```

(You can find the definition of `USBPUMP_HOST_INIT_NODE_VECTOR` in usbkern/host/i/usbpump_host_init.h.)

This structure defines the host init vector, the pre-processing function and the post-processing function. The host init vector initializes a vector of host systems. In the sample applications there is only one host defined. This is generally the case. The pre-processing function does the processing required before the host system(s) are initialized. The post-processing function does the processing required after the host system(s) are initialized. In the sample applications, no pre-processing is required. Host post-processing is almost always required, however.

The initialization of the host init vector for the sample application is below.

```
static
CONST USBPUMP_HOST_INIT_NODE sk_HostInitNodes[] =
          {
          USBPUMP_HOST_INIT_NODE_INIT_V1(
              /* pProbeFn */ NULL,
              /* pUsbdInitFn */ UsbPumpUsbd_Initialize,
              /* pUsbdConfig */ &sk_UsbPumpUsbd_Config,
              /* pUsbdImplementation */ &gk_UsbPumpUsbdImplementation_Minimal,
              /* pUsbdTTInitFn */ UsbPumpUsbdTT_Initialize,
              /* pUsbdTTConfig */&sk_UsbPumpUsbdTT_Config,
              /* pWirelessUsbdInitFn */NULL,
              /* pWirelessUsbdConfig */NULL,
              /* DebugFlags (from USBD_CONFIG) */ 0,
              /* pHcdInitVector */ &gk_UsbPumpHcd_GenericInitVector,
              /* pDriverClassInitVector */ &sk_ClassDriverInitHeader
              )
          };
```

This structure initializes the host system for the sample applications. Here, the initialization functions and structures for the various host system components are assigned. Also, the debug flags and probe function are set for the application. The `pProbeFn`, not used in the sample application, is used to further qualify “which” host systems will be used.

The following sections describe how to set the initialization functions and structures for the various host system components. Note that wireless support is not yet implemented.

4.5.1 Initializing the HCD

To initialize the HCD, point `pHcdInitVector` (in `sk_HostInitNodes`) at `gk_UsbPumpHcd_GenericInitVector`. The `gk_UsbPumpHcd_GenericInitVector` for the sample applications is defined in usbkern/arch/i386/port/catena1650/common/cat1650_hcdconfig.c. Notice the sample applications define one HCD, but you have the option to define as many HCDs as you need.

```
static
__TMS_CONST USBPUMP_HOST_HCD_INIT_NODE HcdInitNodes[] =
          {
          __TMS_USBPUMP_HOST_HCD_INIT_NODE_INIT_V1(
              /* pProbeFn */ NULL,
              /* pInitFn */ isp1761hcd_Init,
              /* pConfig */ &gk_Catena1650_HcdConfig,
              /* DebugFlags */ UDMASK_ANY | UDMASK_ERRORS
              )
          };

__TMS_CONST USBPUMP_HOST_HCD_INIT_NODE_VECTOR gk_UsbPumpHcd_GenericInitVector =
```

```
          __TMS_USBPUMP_HOST_HCD_INIT_NODE_VECTOR_INIT_V1(
              /* name of the vector */ HcdInitNodes,
              /* prefunction */ NULL,
              /* postfunction */ NULL
              );
```

Based on platform/PHY selection, the appropriate `gk_UsbPumpHcd_GenericInitVector` needs to be used.

Please see document *950000324 - MCCI USB DataPump Host Controller* for more details on initializing the HCD.

4.5.2 Initializing the USBD

To initialize the USBD, the following steps need to be followed:

1. Set the pointer to the USBD initialization function
2. Fill in the USBD configuration structure
3. Select a USBD implementation structure
4. Set the pointer to the Transaction Translator initialization function (optional)
5. Fill in the Transaction Translator configuration structure (optional)

4.5.2.1 Setting the Pointer to the USBD Initialization Function

The USBD initialization function (in `sk_HostInitNodes`) should be set to `UsbPump_Usbd_Initialize`, in order to use MCCI's USBD code.

4.5.2.2 Filling in the USBD Configuration Structure

The USBD configuration structure that must be filled in is defined below. The contents of this structure are described in document *950000325 - MCCI USB DataPump Embedded USBDI*.

```
static
CONST USBPUMP_USBDI_USBD_CONFIG sk_UsbPumpUsbd_Config =
          USBPUMP_USBDI_USBD_CONFIG_INIT_V4(
              /* pUsbdName */NULL,
              /* maxNestedCompletions */0,
              /* sizeConfigBuffer */ 0,
              /* sizeSringDescBuffer */0,
              /* maxUrbExtraBytes */0,
              /* tAttachDebounce */USBPUMP_USB20_TATTDB_DEFAULT,
              /* tResetRecovery */USBPUMP_USB20_TRSTRCY_DEFAULT,
              /* tSetAddrCompletion */USBPUMP_USB20_TDSETADDR_DEFAULT,
              /* tSetAddrRecovery */USBPUMP_USB20_TSETADDRRCY_DEFAULT,
              /* tStdRequestNoData */USBPUMP_USB20_TDRQCMPLTND_DEFAULT,
```

```
        /* tStdRequestData1 */USBPUMP_USB20_TDRETDATA1_DEFAULT,
        /* tStdRequestDataN */USBPUMP_USB20_TDRETDATAN_DEFAULT,
        /* tStdRemeRecovery */USBPUMP_USB20_TRSMRCY_DEFAULT,
        /* bNumberHubs*/21,
        /* bPortsPerHub */7,
        /* pHubIdOverrides */NULL,
        /* AnnunciatorMaxSession */4,
        /* ulDebugFlags */ UDMASK_CHAP9 | UDMASK_USBDI | UDMASK_HUB );
```

The sample applications are defined to support 21 hubs with 7 ports per hub, use the standard timeouts defined in reference [USBCORE] and support 4 annunciator sessions. The rest of the fields are not used by the sample application.

4.5.2.3 Selecting a USBD Implementation Structure

There are two pre-defined USBD implementations available to choose from.

1. `gk_UsbPumpUsbdiUsbdImplementation_Minimal`
2. `gk_UsbPumpUsbdiUsbdImplementation_Isoch`

`gk_UsbPumpUsbdiUsbdImplementation_Minimal` indicates minimal support, which includes control transfers, bulk transfers and interrupt transfers, but no isochronous support.

`gk_UsbPumpUsbdiUsbdImplementation_Isoch` indicates isochronous support, which includes control transfers, bulk transfers, interrupt transfers and isochronous transfers.

More information about the USBD implementation choices can be found in document *950000325 - MCCI USB DataPump Embedded USBDI*.

4.5.2.4 Setting the Pointer to the Transaction Translator Initialization Function

The Transaction Translator initialization function (in `sk_HostInitNodes`) should be set to `UsbPumpUsbdTT_Initialize`, in order to use MCCI's USBD code.

4.5.2.5 Filling in the Transaction Translator Configuration Structure

The Transaction Translator configuration structure is defined below. Refer to document *950000548 – MCCI Transaction Translator Users Guide* for more information.

```
static
CONST USBPUMP_USBDI_USBDTT_CONFIG sk_UsbPumpUsbdTT_Config =
        USBPUMP_USBDI_USBDTT_CONFIG_INIT_V1(
            /* fTTPerPort: Allow TTPerPort hub configuration */ TRUE,
            /* bNumberTTs: Calulate number of TTs automatically */ 0
            );
```

In the sample applications, we chose to allow TTPerPort hub configuration and ask the system to calculate the number of TTs supported automatically. In this case, since we decided to allow TTPerPort hub configuration, the number of TTs the system will support is equal to the number of hubs supported multiplied by the number ports per hub supported (21*7). If we chose not to support TTPerPort hub configuration, the number of TTs the system would support would be equal to the number of hubs supported (7).

Carefully select the number of TTs you wish to support, as it will affect RAM usage. You also have the option to over-ride the automatic calculation of number of TTs, by setting bNumberTTs to non-zero.

4.5.3 Adding Support for Class Drivers

In order to define the class drivers to use, the driver class init vector must be filled in. In the sample applications it is defined as:

```
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,
              /* prefunction */ NULL,
              /* postfunction */ NULL
              );
```

Notice that in the sample code, `sk_HostInitNodes` points at this driver class init vector `sk_ClassDriverInitHeader`.

The driver class init vector `(sk_ClassDriverInitHeader)` includes the name of the vector as well as pre-processing and post-processing functions, if desired.

(It is important to recognize the difference between a “driver class” and a “class driver”. A "driver class" is a generic representation of a USB device driver. A "class driver" is an instance of "driver class" for a particular "device class". USB devices are divided into "device classes", such has hub, human interface, etc. The term “class driver instance” refers to an instance of a “class driver”.)

Below is the actual driver class vector definition, as defined in the sample application. Detailed information about how to fill in the individual class driver initialization nodes can be found in the individual class driver user’s guides.

```
static
CONST USBPUMP_HOST_DRIVER_CLASS_INIT_NODE sk_ClassDriverInitNodes[] =
          {
          USBPUMP_HOST_DRIVER_CLASS_INIT_NODE_INIT_V1(
              /* pProbeFn */ NULL,
              /* pInitFn */ UsbPumpUsbdiClassComposite_Initialize,
              /* pConfig */ &sk_UsbPumpUsbdCD_Config.ClassConfig,
              /* pPrivateConfig */ &sk_UsbPumpUsbdCD_Config.ClassPrivateConfig,
```

```
        /* DebugFlags */ UDMASK_ANY | UDMASK_ERRORS
        ),

    USBPUMP_HOST_DRIVER_CLASS_INIT_NODE_INIT_V1(
        /* pProbeFn */ NULL,
        /* pInitFn */ UsbPumpUsbdiClassHid_Initialize,
        /* pConfig */ &gk_UsbPumpUsbdiHid_ClassConfig,
        /* pPrivateConfig */ NULL,
        /* DebugFlags */ UDMASK_ANY | UDMASK_ERRORS | UDMASK_FLOW
        ),

    USBPUMP_HOST_DRIVER_CLASS_INIT_NODE_INIT_V1(
        /* pProbeFn */ NULL,
        /* pInitFn */ UsbPumpUsbdiClassMsc_Initialize,
        /* pConfig */ &gk_UsbPumpUsbdiMsc_ClassConfig,
        /* pPrivateConfig */ NULL,
        /* DebugFlags */ UDMASK_ANY | UDMASK_ERRORS | UDMASK_FLOW
        )
    };
```

The sample applications define three class drivers.

1. Composite
2. Human Interface
3. Mass Storage Class

If you intend on using a MCCI provided class driver, you must set `pInitFn` to the MCCI class driver initialization function, and set the configurations pointer(s) to the MCCI configuration structures. The class driver configuration structures are described in the individual class driver user's guides.

Note that the configuration structures for the composite driver are defined locally in usbkern/host/app/otghidmscapp/otghidmscapp_host_appinit.c (or hosthidmscapp/hosthidmscapp_appinit.c). The configuration structures for HID and MSC are defined in usbkern/host/app/applib/common/hiddemo_create.c and mscdemo_create.c.

Since the composite class driver is included in the vector along with the human interface class driver, the sample applications are able to support a keyboard/mouse composite device. When the keyboard/mouse composite device is plugged in, the USBD will launch an instance of the composite driver. The composite driver will, in turn, launch two HID class driver instances, one for the keyboard and one for the mouse.

4.5.4 Setting the Pointer to the Host Post-Processing Function

You need to assign the pointer to the host post-processing function in the root host structure `gk_UsbPumpHost_GenericInitVector`. In the otghidmscapp.exe sample application, the host post-processing function is named `OtgHidMscApp_Host_InitFinish` and is located in usbkern/host/app/otghidmscapp/otghidmscapp_host_appinit.c. In the hosthidmscapp.exe sample application, the host post-processing function is named `HostHidMscApp_InitFinish` and is located in usbkern/host/app/hosthidmscapp/hosthidmscapp_appinit.c. Note that `OtgHidMscApp_Host_InitFinish` and `HostHidMscApp_InitFinish` are the same.

This post-processing function is executed automatically by `UsbPump_GenericHostInit`, after the HCD, USBD and class drivers are initialized, and the USBD is started.

Once the post-processing is finished, the embedded host/OTG system is up and running and the client application will begin to receive events.

4.5.5 Implementing the Host Post-Processing Function

What the host post-processing function actually does is up to the application designer. But, at a minimum, you will want to create a client object for each class driver you want to receive events from and then register with the class driver in order to begin receiving events. For each instance of a device you want to support, you must register "once". For example, if you want to support a keyboard and a mouse, you would want to register two times with the HID class driver, once for the keyboard and once for the mouse. In the sample application, this is accomplished in `UsbPumpSampleHid_Client_Create` and `UsbPumpSampleMsc_Client_Create`.

Applications never receive events from the Composite Class Driver, so there is no need to register with the Composite Class Driver.

If you want to receive system events from the USBD and/or OTG Annunciator(s), you need to create the notification client objects and open the USBD and OTG Annunciators. In the sample application, this is accomplished in `UsbPumpSampleNotification_Client_Create`, which is located in usbkern/host/app/applib/common/notifcationdemo_create. Please see document *950000325-MCCI USB DataPump Embedded USBDI* and document *950000681-MCCI OTG User's Guide*, respectively, for information on the USBD and OTG Annunciators.

Below is the host post-processing function for the otghidmscapp.exe sample application. The host post-processing function for the hosthidmscapp.exe sample application is identical, but named `(HostHidMscApp_Host_InitFinish)`.

```
static VOID
OtgHidMscApp_Host_InitFinish(
        CONST USBPUMP_HOST_INIT_NODE_VECTOR *    pHostInitHdr,
        USBPUMP_OBJECT_HEADER *      pObjectHeader,
        VOID *                       pUsbdInitContext,
        UINT                         nUsbd
        )
        {
        UPLATFORM * CONST pPlatform = UsbPumpObject_GetPlatform(pObjectHeader);

        USBPUMP_UNREFERENCED_PARAMETER(pHostInitHdr);
        USBPUMP_UNREFERENCED_PARAMETER(pUsbdInitContext);
        USBPUMP_UNREFERENCED_PARAMETER(nUsbd);
```

```
/*
|| Create sample HID client object
*/
UsbPumpSampleHid_Client_Create(
    pPlatform,
    UDMASK_ERRORS | UDMASK_FLOW
    );

/*
|| Create sample MSC client object
*/
UsbPumpSampleMsc_Client_Create(
    pPlatform,
    UDMASK_ERRORS | UDMASK_FLOW
    );

/*
|| Create sample client notification object
*/
UsbPumpSampleNotification_Client_Create(
    pPlatform,
    UDMASK_ERRORS | UDMASK_FLOW
    );
}
```

4.6 Processing Events

Event processing is organized by function; one event handling function for each class driver instance supported. Sample event processing functions are located in usbkern/host/app/applib/common/mscdemo_create.c and hiddemo_create.c. The pointer to each event handler function is assigned when the client object is registered with the particular function (e.g. inside `UsbPumpMsc_Client_Create` or `UsbPumpHid_Client_Create`). The type of events received varies depending on the function. See the particular class driver user's guide for more information.

In addition to receiving events from class drivers, an application may receive system notification events from the USBD and OTG Annunciators. Sample event processing functions are located in notificationdemo_create.c. The pointer to each event handler function is assigned when a session is opened with an Annunciator (e.g. inside `UsbPumpSampleNotification_Client_Create`).

5 Implementing a Custom Class Driver

In most cases, you can use the MCCI provided ClassKit to implement your class driver, which will speed up your development time. For information on how to use the ClassKit, please see document *950000761-MCCI DataPump Embedded Host Class Driver Development Guide*.